ECOLE NATIONALE SUPERIEURE
DES MINES DE SAINT-ETIENNE

N° d'ordre : ID 21

THESE

présentée par

Jacqueline ARGENCE

pour obtenir le titre de :

DOCTEUR

DE L'UNIVERSITE DE SAINT-ETIENNE

ET DE L'ECOLE NATIONALE SUPERIEURE DES MINES DE SAINT-ETIENNE

(Spécialité : Informatique, Image, Intelligence Artificielle et Algorithmique)

ALGORITHMES POUR LE TRACE DE RAYONS DANS LE CADRE D'UNE MODELISATION PAR ARBRE DE CONSTRUCTION

Soutenue à Saint-Etienne le 2 Novembre 1988

COMPOSITION DU JURY

Monsieur C. PUECH		Président et rapporteur
Monsieur J. FRANÇON		Rapporteur
Messieurs	J. AZEMA	Examinateurs
	C. BOUVILLE	
	L-P. UNTERSTELLER	
	B. PEROCHE	
	D. THALMANN	

ECOLE NATIONALE SUPERIEURE
DES MINES DE SAINT-ETIENNE

N° d'ordre : ID 21

THESE

présentée par

Jacqueline ARGENCE

pour obtenir le titre de :

DOCTEUR

DE L'UNIVERSITE DE SAINT-ETIENNE

ET DE L'ECOLE NATIONALE SUPERIEURE DES MINES DE SAINT-ETIENNE

(Spécialité : Informatique, Image, Intelligence Artificielle et Algorithmique)

ALGORITHMES POUR LE TRACE DE RAYONS DANS LE CADRE D'UNE MODELISATION PAR ARBRE DE CONSTRUCTION

Soutenue à Saint-Etienne le 2 Novembre 1988

COMPOSITION DU JURY

Monsieur C. PUECH	Président et rapporteur
Monsieur J. FRANÇON	Rapporteur
Messieurs J. AZEMA	Examinateurs
C. BOUVILLE	
L-P. UNTERSTELLER	
B. PEROCHE	
D. THALMANN	

Je voudrais remercier Monsieur le professeur Claude Puech, directeur du laboratoire d'Informatique de l'Ecole Normale Supérieure, qui m'a fait l'honneur d'accepter de présider le jury de cette thèse.

Je tiens à remercier Monsieur le professeur Jean Françon de l'Université de Strasbourg, rapporteur, dont les remarques m'ont été précieuses.

Monsieur Daniel Thalmann professeur au laboratoire d'Infographie du département d'informatique de l'Ecole Polytechnique Fédérale de Lausanne, a bien voulu accepter un "Rendez-vous à Saint-Etienne", malgré ses nombreuses responsabilités. Qu'il reçoive ici mes plus vifs remerciements.

Monsieur Christian Bouville du CCETT a bien voulu accepter de faire partie du jury. Je tiens particulièrement à le remercier pour toutes ses nombreuses remarques pertinentes.

Je remercie Monsieur Louis Paul Untersteller, directeur de la recherche à Coretech International, pour sa participation à ce jury et pour quelques discussions très intéressantes.

Je remercie Monsieur Jean Azéma, maître de conférence à l'université de Saint-Etienne qui a bien voulu se joindre au jury.

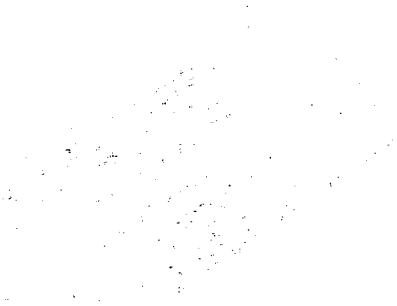
Enfin, je tiens a remercier Monsieur le Professeur Bernard Peroche, directeur du département d'informatique à l'Ecole des Mines de Saint-Etienne, pour m'avoir accueillie au sein de son équipe et pour avoir accepté de diriger ce travail.

Que Messieurs Isambert et Tessier, de Matra, soient remerciés pour leur accueil et leur aide qui m'a permis le portage du logiciel SKY sur leur machine parallèle CAPITAN.

Je remercie tous les membres passés, présents et peut être futurs de l'équipe "communications visuelles", tout particulièrement mon collègue de bureau, Jean Michel Moreau, dont le support moral et linguistique m'a été précieux, Dominique Michelucci pour sa collaboration lors de l'écriture du chapitre sur le tracé de rayons (dans le livre "La synthèse d'images" [PERO88]) dont est issu, remanié, le premier chapitre de cette thèse. Je remercie aussi Michel Beigbeder, Sabine Coquillart, Annie Corbel-Bourgeat, Gilles Fertey, Djamchid Ghazanfarpour, Gabriel Hanoteaux, Ghassan Jahmi, Zhigang Nié et Marc Roelens.

Je voudrais aussi exprimer mon amitié à tous les membres du département. Je remercie l'ensemble du service reprographie de l'Ecole des Mines pour leur diligence qui m'a permis d'avoir ces exemplaires à temps.

Je terminerai en remerciant ma famille, et surtout mon mari, Didier Tallot, qui m'a aidée à surmonter les multiples problèmes rencontrés lors de cette thèse et qui a accepté que je signe cette thèse sous mon nom de jeune fille.



Introduction

En synthèse d'images, le tracé de rayons est incontestablement un des algorithmes qui produisent les images les plus réalistes, à l'heure actuelle. En effet, cette technique permet de rendre les effets de réflexions multiples et de réfraction.

De par sa simplicité, une méthode de tracé de rayons est introduite dès 1968 par APPEL [APPE68], et une implantation dans un système de CAO-CFAO est réalisée dès 1971, par GOLDSTEIN et NAGEL [GOLD71] du MAGI (Mathematics Applications Group,Inc.). Le principe de cette première réalisation est de trouver quel est l'objet vu en chaque pixel de l'écran en calculant les points d'intersection de chaque objet de la scène avec la demi-droite (le "rayon") issue de l'œil et passant par le centre du pixel calculé. Le point d'intersection le plus proche de l'œil donne l'objet vu. Ensuite, le pixel est colorié avec la couleur de cet objet. Cette méthode simple est extrêmement brutale, d'où son autre nom de "brute force algorithm" dans l'article de SUTHERLAND et al. [SUTH74]. Les machines suffisamment puissantes pour générer une image en un temps raisonnable avec cette technique sont bien rares dans les années 1970 ; aussi cette méthode est-elle plus ou moins abandonnée jusqu'en 1980, date à laquelle WHITTED publie "An improved illumination model for shaded display" [WHIT80] ; cet article montre plusieurs images présentant des effets optiques inédits de réflexion et d'ombrage avec des objets opaques ou transparents. L'algorithme, simple et général, prend en compte la plupart des effets optiques : réflexions multiples, transparences, réfractions, sources lumineuses multiples. C'est sans doute là, la véritable naissance du tracé de rayons ("ray casting" ou "ray tracing" dans la littérature anglaise).

En 1982, ROTH [ROTH82] propose différents moyens d'accélérer l'algorithme, et étend le champ d'application du tracé de rayons aux objets modélisés par un arbre de construction ou arbre CSG et à la visualisation interactive de type fil de fer. La même année, le groupe MAGI utilise le tracé de rayons pour réaliser des images de synthèse du film "Tron" [SORE82]. Depuis, d'autres films d'animation ont été réalisés aussi grâce au tracé de rayons ("Quest"[1], [LEIS88][2], ...) et les publications dans ce domaine sont nombreuses.

---

[1] film réalisé sur un réseau de stations de travail Apollo
[2] film réalisé sur un réseau de stations de travail Sun

Dans le premier chapitre de cette thèse seront exposées les bases du tracé de rayons et un rapide panorama des développements apportés. Les chapitres suivants seront consacrés aux recherches que nous avons faites pour améliorer le tracé de rayons dans le cadre d'un environnement de modélisation par arbre de construction (ou arbre CSG). La définition d'un arbre de construction est présentée dans le premier chapitre. Le chapitre 2 aborde plusieurs solutions originales aux problèmes d'optimisation du tracé de rayons. Ce chapitre contient aussi une extension du modèle CSG résolvant les conflits amenés par deux objets occupant le même espace ou possédant une face commune. Toujours dans le but d'améliorer les performances de l'algorithme du tracé de rayons dans le cadre d'une modélisation CSG, nous définirons de nouvelles primitives. Le chapitre 3 est dédié à une nouvelle méthode efficace d'antialiassage du tracé de rayons.

Les mémoires d'images permettant d'afficher en vraies couleurs (c'est à direqui possèdent plus de 24 bits par pixels pour coder la couleur) sont encore trop chères pour être courantes. Beaucoup de stations de travail ne peuvent afficher que 256 couleurs parmi 16 millions. Le choix de ces 256 couleurs est crucial pour ne pas trahir la qualité du rendu de l'algorithme du tracé de rayons. Le quatrième chapitre repose sur un algorithme de quantification des couleurs par un octree utilisant le balayage de Peano.

Le dernier chapitre décrira de façon plus précise le logiciel de tracé de rayons : SKY, qui intègre les résultats des chapitres 2, 3 et 4. La parallélisation sur une machine de type MIMD (Multiple Instruction Multiple Data): CAPITAN[3] est finalement exposée.

En annexe, les résultats, temps de calcul et statistiques sur certaines images tests seront regroupés. D'autre part, nous avons essayé d'établir une bibliographie complète sur le tracé de rayons. Seuls les travaux qui se rapprochent des sujets de cette thèse ont été cités dans le texte. Nous avons jugé utile, cependant, de conserver toutes les références au sein de notre bibliographie.

---

3 © Matra

Chapitre 1

Présentation générale de l'algorithme du tracé de rayons

1. Présentation du tracé de rayons

Depuis 1980, de nombreuses équipes, en quête d'un plus grand réalisme de l'image synthétique, ont exploré l'univers du tracé de rayons ([KAJI83]). Ce chapitre se veut un panorama, parfois rapide et pas toujours exhaustif, des recherches sur le thème du tracé de rayons. Ce chapitre se découpe en six parties regroupant les sujets suivants:

- le principe de l'algorithme.
- les calculs d'intersection avec divers types d'objets.
- les optimisations logicielles
- la prise en compte de certains effets optiques, images fil de fer, l'antialiassage.
- la parallélisation de l'algorithme.

1.1. Principe de l'algorithme

Au départ, le tracé de rayons se comporte comme la méthode de la "force brutale" ; il calcule l'image pixel par pixel ; pour déterminer l'objet vu en un pixel, les points d'intersection du rayon œil-pixel avec tous les objets de la scène sont calculés, et le plus proche de l'œil est retenu : P ; ce rayon que l'on appelle primaire (issu de l'œil) a été lancé dans la scène pour trouver l'objet vu en un pixel.

Supposons l'existence d'une source lumineuse dans la scène; la couleur de l'objet O vu au point P dépend de la couleur propre de O en ce point, mais aussi de son éclairement : P est-il éclairé par la source, ou est-il masqué par un autre objet ? Pour le savoir, l'algorithme lance un rayon secondaire dans la scène. Ce rayon est issu du point P et dirigé vers la source lumineuse ; son intersection avec tous les objets de la scène est testée; s'il rencontre un objet opaque situé entre P et la source, alors P n'est pas éclairé par la source, mais seulement par la lumière ambiante : c'est une loi simple de l'optique géométrique. L'algorithme peut ainsi prendre en compte plusieurs sources lumineuses : il suffit de lancer autant de rayons secondaires qu'il y a de sources lumineuses dans la scène, et d'ajouter la contribution des éclairements au point P.

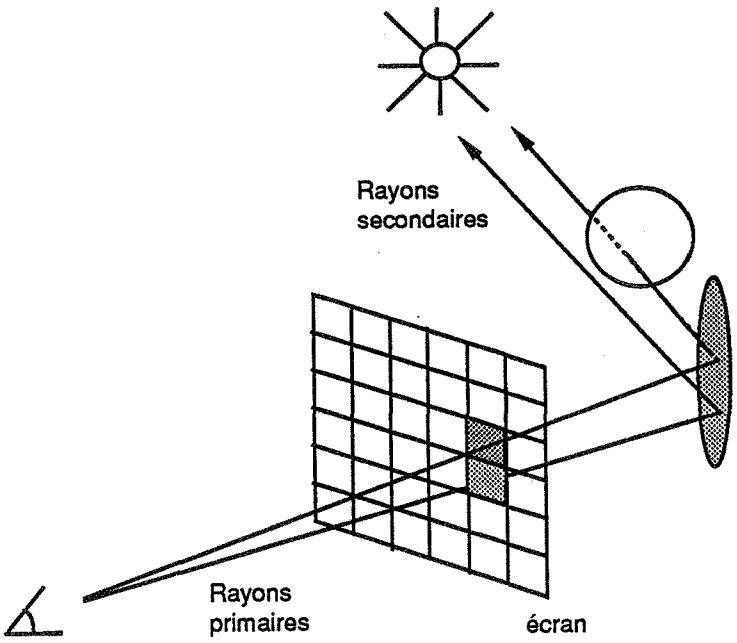


Fig. 1 Principe du tracé de rayons

Récapitulons schématiquement l'algorithme :

```
POUR chaque pixel de l'écran
    définir le rayon primaire œil-pixel ;
    POUR chaque objet
        tester les intersections ;
        prendre la plus proche ;
        relancer des rayons vers les sources lumineuses ;
        calculer l'éclairement du point d'intersection ;
        affecter la couleur au pixel ;
    FPOUR
FPOUR
```

Mais d'autres phénomènes peuvent influer sur la couleur du pixel; par exemple, si l'objet O vu en P a une surface réfléchissante (miroir, eau, métal, etc) alors la couleur du pixel dépend de la couleur de l'objet réfléchi en P. Pour déterminer quel est l'objet qui se reflète en P, un rayon secondaire "réfléchi" est lancé dans la scène; ce rayon est issu de P, et sa direction est symétrique de celle du rayon incident en P (ici le rayon primaire) par rapport à la normale de l'objet en P : c'est une des lois de l'optique géométrique de Descartes (voir paragraphe 4.1). Après avoir testé l'intersection de tous les objets de la

scène avec le rayon réfléchi, l'algorithme connaît l'objet qui se reflète en P; c'est O', et le point de O' qui se réfléchit en P est P'. La couleur de P' ne dépend pas seulement de O', mais aussi de l'éclairement de P' : est-il éclairé ou non ? Un rayon tertiaire vers la ou les sources lumineuses est lancé.... etc. Le processus finira par s'arrêter : les rayons transportent de moins en moins d'énergie lumineuse et au bout d'un certain temps (parfois d'un temps certain !) il devient inutile de lancer d'autres rayons.

Revenons au pixel, en lequel est vu le point P de l'objet O; peut-être O est-il transparent, ou translucide ? En ce cas la couleur de P dépend aussi de la couleur de l'objet que l'on voit à travers O au point P; quel est-il ? Pour le savoir, l'algorithme lance un rayon secondaire réfracté à partir de P; la direction de ce rayon secondaire dépend de la direction du rayon d'incidence (ici le rayon primaire) et de l'indice de réfraction de O, selon une autre loi bien connue de l'optique géométrique (voir paragraphe 4.1).

Le tracé de rayons suit donc à l'envers le parcours des rayons lumineux.

1.2. Premiers constats

Cette première formulation de l'algorithme permet de dresser quelques constats essentiels :

- le tracé de rayons prend en compte de nombreux phénomènes optiques, puisqu'il imite la nature (ou plutôt un modèle physique de la nature !) en suivant à l'envers les rayons lumineux. D'où le réalisme des images qu'il produit.

- le tracé de rayons est une méthode extrêmement simple. Les seuls calculs géométriques nécessaires sont ceux de l'intersection entre une demi-droite et un objet.

Ce dernier problème est manifestement plus facile que celui de l'intersection entre deux surfaces gauches, ou entre deux objets quelconques. De plus, si la scène compte n types d'objets (faces, quadriques, bicubiques, etc), le tracé de rayons nécessite seulement n procédures, parfaitement indépendantes; ainsi, s'il devient nécessaire d'ajouter un autre type d'objet dans un logiciel de tracé de rayons, il suffit d'ajouter la procédure correspondante, qui sait calculer l'intersection entre "son objet" et un rayon, et le reste du logiciel reste inchangé.

Les inconvénients de cette méthode apparaissent tout aussi clairement : l'image présente des défauts d'aliassage. L'aliassage concerne d'abord les rayons primaires : certains bords apparaissent crénelés sur l'image; des petits objets peuvent être omis s'ils passent entre les pixels; des objets longs et minces peuvent ainsi apparaître puis disparaître.

En résumé, un rayon primaire par pixel n'est donc pas toujours suffisant. Mais la même remarque peut être faite pour les rayons secondaires (cas des réflexions et lumières diffuses). Certes, ce problème d'aliassage a un remède très simple : lancer plus de rayons.

Mais alors on se heurte au deuxième problème du tracé de rayons : cette méthode exige une quantité colossale de calculs, bien supérieure à celles des méthodes antérieures (il est vrai qu'elle produit de plus belles images).

On peut évaluer ainsi le nombre de tests d'intersection : une image compte p ($\approx 10^6$) pixels; en comptant un rayon primaire par pixel, une moyenne de deux rayons secondaires par pixel (la prise en compte des lumières diffuses en exige bien plus), et en supposant que la scène compte n objets élémentaires, alors $3pn \approx 3\ 10^6$ n tests sont nécessaires; c'est ce qui explique que les premières images produites par le tracé de rayons aient été si simples, la scène visualisée ne comptant souvent qu'une dizaine d'objets élémentaires. Pour l'instant, malgré le progrès des performances matérielles, les coûts de calcul sont encore la principale limite du tracé de rayons.

2. Définitions

2.1. Arbre de construction

Un arbre de construction (en anglais C.S.G., Constructive Solid Geometry)[1] est décrit de la façon suivante [REQU80]:

- un nœud est une opération ensembliste du type réunion, intersection ou différence,
- une feuille est un objet élémentaire comme : une sphère, un cube, un cône, un cylindre ...

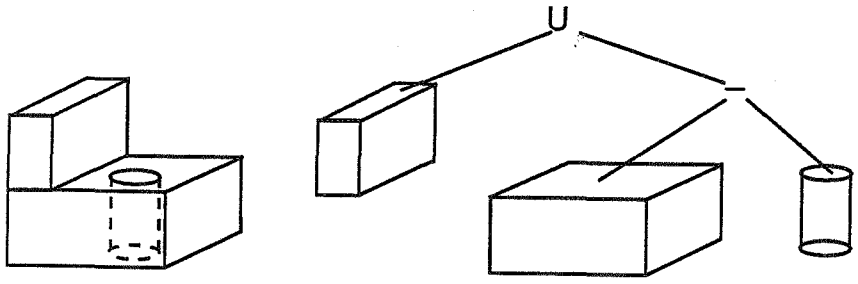


Fig. 2 Exemple d'arbre de construction

[1] L'abréviation CSG est couramment utilisée tant dans la littérature anglaise que française. Pourtant, diverses traductions ont été proposées : arbre de construction, arbre de représentation qui ont l'inconvénient de ne pas avoir une corrélation évidente avec le terme CSG, et géométrie solide constructive qui n'est guère naturelle. Nous proposons une francisation du terme CSG qui permet de conserver cette abréviation en opérant un léger glissement de sens : Construction de Solides Géométriques. Dans toute cette thèse nous utiliserons indifféremment les termes arbre de construction et CSG.

REMARQUE

Dans toute la suite de cette thèse, on définira tout point par ses coordonnées homogènes cad :

> si (x,y,z) sont les coordonnées d'un point M dans un repère R, ses coordonnées homogènes seront (X,Y,Z,T) avec :
> $x = \frac{X}{T}$, $y = \frac{Y}{T}$, $z = \frac{Z}{T}$ et $T \neq 0$,
>
> un vecteur $\vec{V}$ (v_x,v_y,v_z) aura pour coordonnées homogènes :
> $(\lambda v_x,\lambda v_y,\lambda v_z,0)$ avec $\lambda \neq 0$, si $\vec{V} \neq \vec{0}$.

Cette notation permet d'écrire une translation t (t_x, t_y, t_z) sous forme matricielle :

$$\begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Pour simplifier la description de l'arbre de construction, une primitive est définie dans un repère local dans lequel, par exemple, la primitive cube est un cube de coté 1 dont l'un des sommets est l'origine de ce repère et qui se trouve entièrement dans la partie positive de l'espace.

Une feuille sera donc composée d'une primitive et d'une matrice de transformation qui appliquée à tous les points de la primitive (volumique) donnera l'objet élémentaire. Cette matrice M, transforme le repère local de la primitive en repère de la scène (ou repère global). Notons $R=(O, \vec{i}, \vec{j}, \vec{k})$ le repère de la scène et $R'= (O, \vec{i'}, \vec{j'}, \vec{k'})$le repère local de la primitive. On a, par conséquent :

$O = MO'$, $\vec{i} = M\,\vec{i'}$, $\vec{j} = M\,\vec{j'}$, $\vec{k} = M\vec{k'}$.

Par exemple (cube, $\begin{pmatrix} 2 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$) représente l'objet :

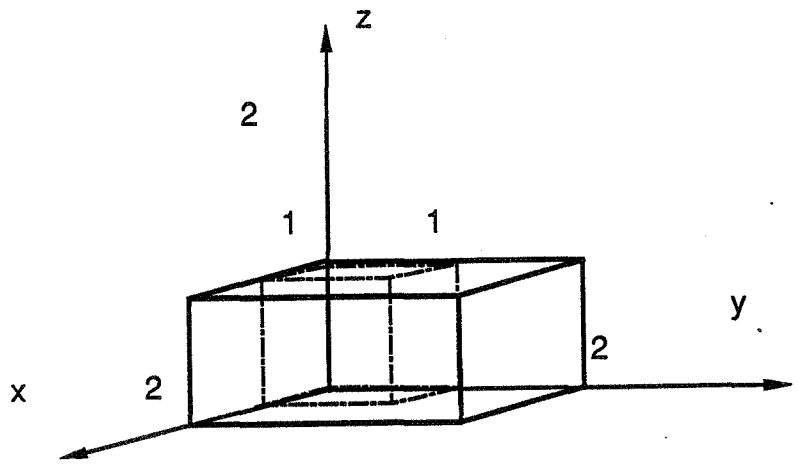


Fig. 3 Passage du repère local au repère de la scène.

2.2. Quadtree , bintree et octree

Un quadtree (ou arbre quadrant[2]) est un arbre dont tous les noeuds possèdent quatre fils, de plus cet arbre représente une région rectangulaire du plan subdivisée récursivement en quatre parties égales, si une certaine condition n'est pas vérifiée. Ce qui veut dire qu'à chaque nœud est associée une région de l'espace divisée en quatre parties égales chacune associée à l'un de ses quatre fils. Prenons comme exemple la représentation par un quadtree d'une image noire et blanche :

[2] Nous n'avons pas trouvé d'autre proposition de traduction. A notre sens, quadrant permet de suggérer la notion de division de l'espace en quatre parties égales. Cela nous permet de garder une cohérence avec la traduction d'octree par arbre octant. La traduction d'octree par arbre octal ne donne que la notion de huit fils sans associer l'idée de division de l'espace en huit parties égales. De plus un équivalent pour quadtree n'est pas évident ("arbre quadral"?).

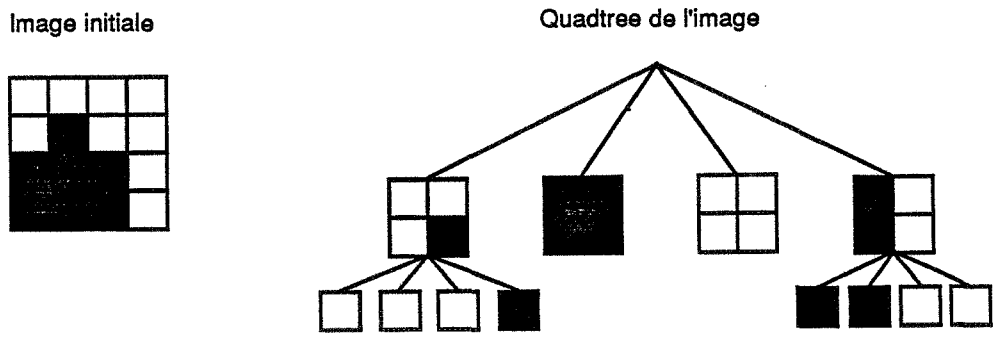


Fig. 4 Représentation par un quadtree d'une image noire et blanche

Un bintree (ou arbre dichotomique)[3] est un arbre binaire qui sépare en deux l'espace en alternance dans les deux directions (en 2 D), dans les trois (en 3D). Si nous reprenons l'exemple précédent :

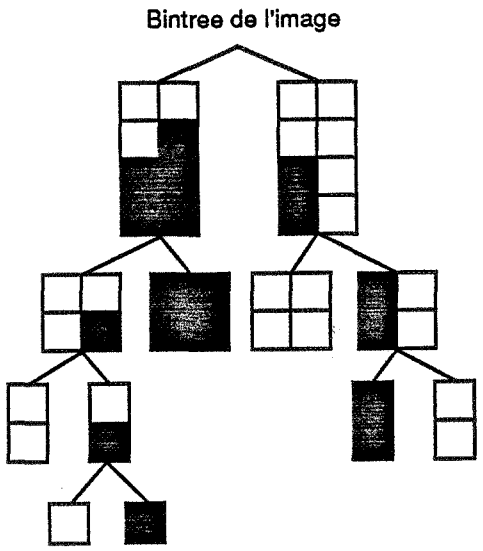


Fig. 5 Représentation par un bintree de l'image noire et blanche

Un octree (ou arbre octant) est l'extension d'un quadtree en 3D. L'espace est divisé, récursivement, en huit parallélépipèdes.

[3] Nous proposons cette traduction qui suggère bien le découpage récursif en deux. Pour garder une certaine cohérence avec les quadtrees et les octrees il faudrait inventer de nouveaux mots : arbre quadritomique et arbre octotomique ? L'histoire jugera !

3. Calculs d'intersections

Pour chaque pixel, le tracé de rayons effectue en gros deux types de calculs : des calculs géométriques d'intersection entre un rayon et des objets d'une part, d'autre part des calculs optiques (opérations de mélange de couleurs), quand les impacts sont connus. Les plus coûteux sont les calculs géométriques, d'autant plus que leur volume croît avec la complexité de la scène, alors que les calculs optiques (mélange des couleurs) restent proportionnels au nombre des points d'intersection. Les procédures qui calculent l'intersection entre un rayon et un objet sont donc d'une importance cruciale : de leur efficacité dépend celle du tracé de rayons. Classiquement, à chaque type d'objet (faces planes, sphères, cylindres, cônes, autres quadriques, tores, bicubiques, objets composites, objets fractals, etc) est attachée une procédure, qui calcule l'intersection entre cet objet et un rayon.

Nous détaillons dans ce qui suit les procédures d'intersection avec les objets les plus courants. Certains de ces objets ne sont pas indispensables; ainsi les polyèdres pourraient être décrits par des objets composites (obtenus par des opérations booléennes); mais cette description ne permettrait pas de leur attacher un bon "volume englobant", technique d'optimisation la plus simple du tracé de rayons. En effet, si on laisse l'utilisateur libre d'employer des demi-espaces infinis, la scène à visualiser peut comporter un objet infini n'ayant pas de volume englobant fini et par conséquent peu efficace.

3.1. Intersection avec un arbre CSG

3.1.1. Intersection avec une instance d'objet

On utilise souvent des repères locaux aux objets; ceci permet de factoriser les descriptions d'objets les plus encombrantes, et de calculer les points d'intersection dans les repères les plus commodes; attacher un repère à un objet permet aussi de spécifier assez facilement sa position dans le monde. A chaque instance[4] d'un objet dans la scène est attachée une matrice de transformation linéaire 4x4, composée de rotations, translations, affinités et symétries : c'est la matrice de passage du repère global ou repère de la scène (le monde) au repère local (le repère dans lequel est défini l'objet). Une "primitive" est un objet unitaire dans son repère local; le cube unité $[0..1]^3$, la sphère unité de centre (0 0 0 1) et de rayon 1 sont des primitives classiques; moyennant les transformations géométriques, le cube unité peut donner naissance à tous les parallélépipèdes, et la sphère à tous les ellipsoïdes.

Deux possibilités s'offrent pour calculer l'intersection d'une instance (objet, matrice M) et d'un rayon : soit l'objet est transformé dans le repère de la scène, soit le rayon est

[4] L'emploi de la terminologie des langages objet n'est pas un hasard.

transformé dans le repère local de l'objet. Evidemment, il est plus simple et plus rapide de transformer la demi-droite du rayon plutôt qu'un objet d'une complexité arbitraire. Le rayon est défini dans le repère global par son extrémité $E=(E_x\ E_y\ E_z\ 1)$ et son vecteur directeur $\vec{V}=(V_x\ V_y\ V_z\ 0)$; autrement dit, le rayon est la demi-droite $P(t) = t\ \vec{V} + E$ et $t > 0$. Transformé dans le repère local, le rayon devient $P'(t)= P(t)M = (E+t\ \vec{V})M$; il a donc pour extrémité $E'=EM$ et pour vecteur directeur $\vec{V}' = \vec{V}\ M$. Pour transformer le rayon, il suffit donc de multiplier E et $\vec{V}$ par la matrice M de passage du repère global au repère local de l'objet; la paramètrisation du rayon est indépendante du repère.

Quand la scène est décrite par une hiérarchie d'objets, un arbre de construction par exemple, il serait maladroit de conserver des transformations géométriques à tous les niveaux : transformer un rayon dans le repère local d'un objet de profondeur n nécessiterait alors $O(n)$ multiplications matricielles. Aussi, pour améliorer les performances, un prétraitement descend-il toutes les matrices au niveau des feuilles de l'arbre, les primitives. Ainsi, transformer un rayon dans le repère local d'une primitive n'exige plus qu'une seule multiplication matricielle sur l'extrémité et le vecteur directeur du rayon.

Le tracé de rayons a aussi besoin de la normale aux points d'intersection. Soit $P'=\begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix}$ les coordonnées d'un point d'intersection dans le repère local de l'objet; la procédure calculant le point d'intersection P' peut aussi rendre $\begin{pmatrix} a' \\ b' \\ c' \\ 0 \end{pmatrix}$, le vecteur normal en P'; posons $d' = -(a'x'+b'y'+c'z')$ et $R' = \begin{pmatrix} a' \\ b' \\ c' \\ d' \end{pmatrix}$; $P'R'=0$ est l'équation du plan tangent en P', exprimée dans le repère local. Soit $P=\begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$ les coordonnées du point d'intersection dans le repère global On détermine l'équation du plan tangent en P dans ce repère de la manière suivante : $P'=PM$ et $P'R'=0$ impliquent trivialement $P(MR')=0$, ce qui est l'équation du plan tangent en P : il est donc possible de "sortir" la normale du repère local.

3.1.2. Intersection avec un objet composite

Le tracé de rayons peut accepter en entrée une scène, ou des objets, décrits par des arbres de construction. La prise en compte des opérations ensemblistes ne lui fait rien perdre de son élégance et de sa simplicité. Ce paragraphe expose comment réduire récursivement le problème de l'intersection entre un rayon et un arbre de construction à celui de l'intersection entre un rayon et un objet élémentaire.

Soit à calculer l'intersection entre un solide décrit par un arbre A et un rayon R. L'arbre A est binaire; les nœuds de A sont des opérateurs ensemblistes : l'intersection, notée $\cap$, la réunion, notée $\cup$, et la différence, notée -. Les feuilles de A sont les objets élémentaires, des instances de primitives.

Si A est une feuille, alors l'intersection entre R et A est calculée en appelant une des procédures spécialisées, détaillées plus bas. Cette procédure rend une liste d'intervalles d'intersection ($[t_1,t_2]$, $[t_3,t_4]$, ...) le long de R; cette liste est ordonnée par t_i croissant; elle peut bien sûr être vide.

Si A est un nœud, soient % l'opérateur ensembliste de A, G son fils gauche, et D son fils droit : A = G % D. Alors les intersections entre R et G, et entre R et D sont calculées par un appel récursif; les résultats sont deux listes d'intervalles : LG et LD. L'intersection entre R et A est LG % LD : il faut donc savoir effectuer une opération ensembliste sur deux listes d'intervalles, ce qui est un exercice simple et classique de programmation.

Quelques optimisations sont possibles; si A porte l'opération ensembliste $\cap$ et que l'intersection entre R et G est vide, alors il est inutile de calculer l'intersection entre R et D : de toutes façons, l'intersection entre R et A= G $\cap$ D est vide; enfin il vaut mieux chercher d'abord l'intersection qui se calcule le plus rapidement. De même si l'opérateur de A est -, et si l'intersection entre R et G est vide : alors forcément l'intersection entre R et A= G - D est vide.

3.1.3. Intersection avec une primitive

3.1.3.1. Le cube unité

La méthode naïve calcule tous les points d'intersection avec les six faces du cube, vérifie l'appartenance du point d'intersection trouvé à la face considérée et prend la plus proche et la plus éloignée des intersections[5].

[5] Quand le rayon intersecte une des arêtes du cube, on peut avoir plus de deux intersections.

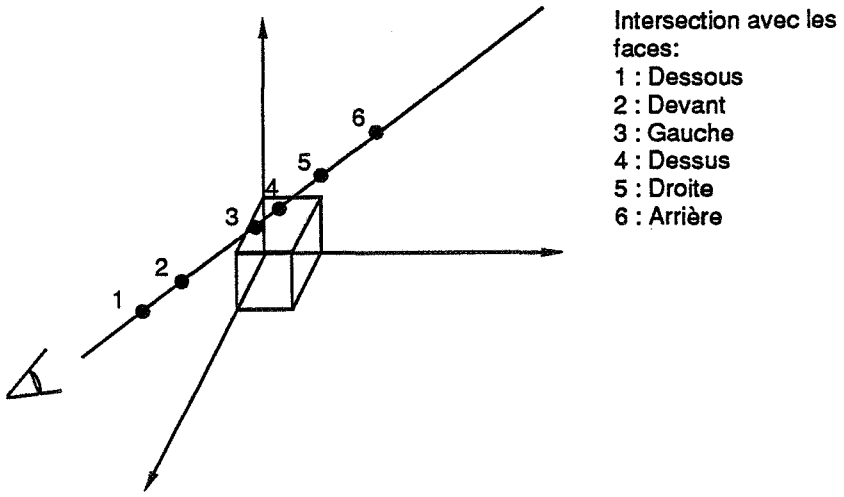


Fig. 4 Intersection d'un rayon avec les plans définissant un cube

En réalité, on peut améliorer ces performances en stoppant le calcul des intersections avec les faces dès que l'on a trouvé deux points d'intersection (s'ils ne sont pas confondus à un epsilon près). Dans le pire des cas, ce n'est pas meilleur mais en moyenne c'est plus rapide.

3.1.3.2. La sphère unité

La sphère unité est centrée à l'origine et a pour rayon 1; son équation est :

$$f(x,y,z) = x^2 + y^2 + z^2 - 1 = 0 \qquad (1).$$

Transformé dans le repère local de la sphère, le rayon devient E+tV, E=(Ex Ey Ez 1) et $\vec{V}$=(Vx Vy Vz 0). Après substitution des valeurs de x,y,z dans (1), une équation en t du second degré est obtenue : $At^2+2Bt+C=0$, avec $A= \vec{V}\,{}^t\vec{V}$, $B= \vec{V}\,{}^tE$ et $C= E\,{}^tE - 1$. Si $D = B^2 - AC$ est négatif ou nul, alors il n'y a pas d'intersection. Sinon il y a intersection pour

$$t = \frac{(-B \pm \sqrt{D})}{A}.$$

La normale en un point (x y z) de la sphère est $(f_x'\ f_y'\ f_z') = (2x\ 2y\ 2z)$.

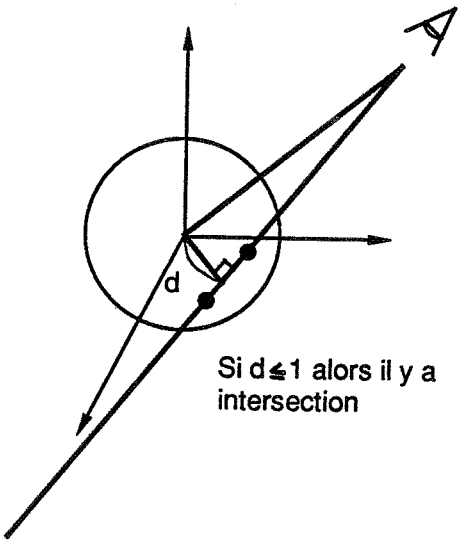


Fig. 5 Intersection d'un rayon et d'une sphère

Calculer les points d'intersection d'un rayon et d'une sphère est facile et rapide : c'était l'objet favori des premières images réalisées en tracé de rayons. Pour la même raison, les sphères sont souvent utilisées comme volume englobant, comme nous le verrons ultérieurement. Les autres procédures d'intersection ne seront pas aussi détaillées.

3.1.3.3. Le cône unité

L'équation d'un tronc de cône dans son repère local est la suivante :

$$(x^2 + y^2 - z^2 + 2z - 1 = 0) \text{ et } (0 \leq z \leq 1).$$

La base est le cercle horizontal de centre O et de rayon 1, le sommet est en (0 0 1). L'intersection est triviale : soit c l'intervalle intersection avec le cône infini, et s l'intervalle intersection avec $0 \leq z \leq 1$; ils sont éventuellement vides; le résultat est $(c \cap s)$. En un point (x y z 1) du cône, avec z non nul, la normale est (2x 2y -2z+2 0). La primitive "tronc de cône" n'est théoriquement pas indispensable, un objet composite suffirait (un cône infini moins un demi-espace). Mais cela ne permettrait pas d'attacher au tronc de cône un volume englobant efficace.

3.1.3.4. Le cylindre unité

L'équation d'un cylindre dans son repère local est : $(x^2 + y^2 - 1 = 0)$ et $(0 \leq z \leq 1)$. La base du cylindre est le cercle horizontal de centre O et de rayon 1. Comme pour le cône, la primitive "tronc de cylindre" n'est théoriquement pas indispensable, et un objet composite suffirait; mais cela ne permettrait pas d'attacher au tronc de cylindre un volume englobant efficace. Le calcul de l'intersection et de la normale est trivial.

3.1.3.5. Intersection avec une quadrique quelconque

Nous avons déjà étudié le cas de certaines quadriques dans les paragraphes précédents; les quadriques suivantes sont moins employées : paraboloïde hyperbolique, hyperboloïde à une nappe, hyperboloïde à deux nappes. Dans ces cas, une procédure générale peut être préférée à un chapelet de procédures particulières.

Une quadrique a pour équation générale :

$$f(x,y,z) = Ax^2 + By^2 + Cz^2 + 2Dxy + 2Exz + 2Fyz + Gx + Hy + Iz + J = 0 \quad (2).$$

Pour calculer l'intersection, x, y, z sont substituées par leur valeur :

$$(x\ y\ z\ 1) = E + t\ \vec{V}.$$

Une équation algébrique en t, du second degré, est obtenue et résolue. Le vecteur normal en (x y z 1) est $(f'_x\ f'_y\ f'_z\ 0)$, soit

$$(2Ax + 2Dy + 2Ez + G,\ 2Dx + 2By + 2Fz + H,\ 2Ex + 2Fy + 2Cz + I,\ 0).$$

3.1.3.6. Un tore

Dans son repère local, le tore s'obtient par la rotation autour de l'axe Oz du cercle vertical de rayon r et de centre (1 0 0). Le tore a pour équation :

$$(x^2 + y^2 + z^2 - r^2 + 1)^2 - 4(x^2 + y^2) = 0.$$

En substituant x, y et z par leur valeur : (x y z 1)= E+Vt, une équation en t du quatrième degré est obtenue; elle peut être résolue par des méthodes numériques ou analytiques.

3.1.3.7. Un polygone convexe

Un polygone convexe dans le plan Oxy est défini par un système de n inéquations linéaires : $a_i x + b_i y + c_i \leq 0$.

Le rayon est transformé dans le repère local du polygone. Il n'y a pas intersection quand le rayon R, est parallèle au plan Oxy : $V_z = 0$ ou quand R appartient à Oxy. Sinon, l'intersection de R avec Oxy est donné par : $z = E_z + tV_z = 0$. On en déduit les valeurs de t, x et y du point $R \in$ Oxy. Encore faut-il que ce point soit à l'intérieur du polygone; ceci est vrai s'il vérifie le système $a_i x + b_i y + c_i \leq 0$ (i de 1 à n). Ce test est en $O(n)$; Il existe un algorithme en $O(\log(n))$, moins simple exposé dans [PERO88].

Les rectangles et les parallélogrammes sont très employés dans certaines scènes; ce sont tous des instances du carré unitaire. Les calculs sont alors encore plus simples : le point $R \in$ Oxy est à l'intérieur du carré si et seulement si $0 \leq x \leq 1$ et $0 \leq y \leq 1$.

3.1.3.8. Un polygone quelconque

Le polygone est décrit par n arêtes orientées dans le plan Oxy; son contour n'est pas auto-intersectant, si bien que l'intérieur peut être défini par parité.

Le rayon est transformé dans le repère local du polygone. L'intersection entre le rayon et le plan Oxy est calculé. Puis on vérifie que ce point d'intersection, s'il existe, se trouve bien à l'intérieur du polygone. Les algorithmes les plus simples sont en O(n), n étant le nombre d'arêtes du polygone; en utilisant des structures de données plus sophistiquées et un ordre sur les segments, une solution en O(log(n)) est possible. Dans un contexte similaire, Kajiya ([KAJI83]) propose l'utilisation d'un strip-tree. Cependant, n doit être assez grand pour que ces considérations théoriques deviennent pertinentes.

3.1.3.9. Un prisme

Le prisme est situé entre les plans z=0 et z=1; son axe est vertical; sa base est un polygone; bien sûr ce polygone peut approcher des arcs de courbe avec assez de précision, s'il compte suffisamment de côtés. Kajiya calcule ainsi l'intersection entre le rayon et le prisme :

Le rayon est transformé dans le repère local du prisme, où il a pour extrémité E et pour vecteur directeur V. On peut écarter les cas particuliers suivants : le rayon est horizontal et plus haut ou plus bas que le prisme; le rayon n'intersecte pas un éventuel volume englobant le prisme; dans le cas où le rayon est vertical, le problème se réduit à un test d'inclusion d'un point dans la base du prisme;

Dans le cas général, l'intersection est d'abord étudiée en projection sur le plan z=0; la projection sur z=0 du rayon a pour extrémité $E'=(E_x\ E_y\ 0\ 1)$ et pour vecteur directeur $\vec{V}'=(V_x\ V_y\ 0\ 0)$. Les intersections avec les segments de la base du prisme sont déterminées; s'il n'y a aucun point d'intersection, le rayon n'intersecte pas le prisme; dans le cas contraire, ces points d'intersection sont classés, du plus proche au plus éloigné de E', grâce à la paramétrisation du rayon. Les deux points d'intersection entre le rayon et les plans z=0 et z=1 sont aussi insérés en bonne place dans cette liste.

L'algorithme "remonte" ensuite les points d'intersection sur le rayon. Un point de paramètre t est la projection du point $E+t\vec{V}$ situé sur le rayon; sa cote est donc $z=E_z+tV_z$. Ce point est situé sur une face verticale du prisme quand $0 < z < 1$; ce point est situé sur une des deux faces horizontales du prisme quand (z=0 ou z=1) et que le point est à l'intérieur de la base du prisme. Un parcours de la liste ordonnée permet alors de déterminer le premier point d'intersection, s'il existe. Pour les opérations ensemblistes, il est aussi possible de rendre une liste d'intervalles d'intersection.

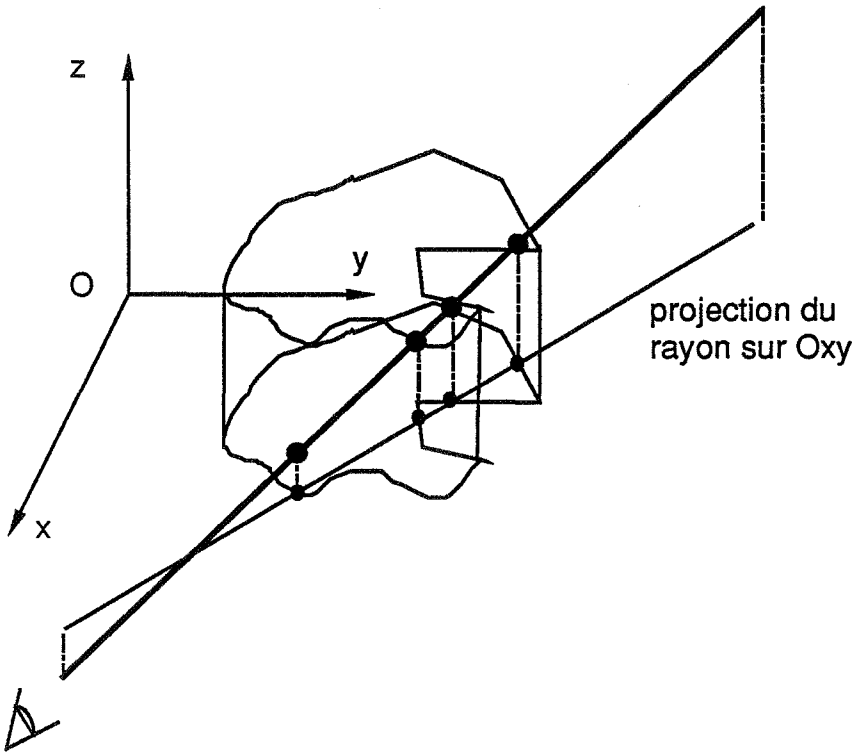


Fig. 6 Intersection entre un rayon et un prisme

Kajiya [KAJI82] propose de décrire le polygone base du prisme par un "strip-tree". Rappelons brièvement qu'un strip-tree est un arbre binaire de rectangles englobant une courbe ou une ligne brisée complexe; les feuilles du strip tree sont des segments, ou des rectangles suffisamment plats pour être considérés comme des segments; cette représentation accélère les calculs d'intersection pour les prismes à base compliquée. Les strip-trees permettent de calculer des normales "raisonnables" aux courbes qu'ils représentent.

3.1.3.10. Intersection avec un polyèdre convexe

Un polyèdre convexe est défini par n inéquations linéaires

$$e_i(x,y,z) = a_i x + b_i y + c_i z + d_i > 0.$$

Comme par ailleurs (x y z) doit se trouver sur le rayon d'extrémité E et de vecteur $\vec{V}$, (x y z 1) = E + t $\vec{V}$, on en déduit trivialement n inéquations $e_i(t)>0$. L'intersection entre le rayon et le polyèdre convexe est soit l'intervalle vide, soit un intervalle $]t_0, t_1[$, avec $t_0 < t_1$, qui peut se calculer ainsi, en O(n) :

```
fonction intersection_avec_polyèdre
   solution = ]0, +∞[  /* le rayon lui même */ ;
   POUR i de 1 à n
      solution = solution ∩ (]0,+∞[ ∩ (ei(t)>0) );
      SI ( solution=∅ ) ALORS rendre( ∅); FSI
   FPOUR
   rendre (solution);
```

L'opération ensembliste ∩ sur deux intervalles est triviale. Le lecteur aura reconnu dans cet algorithme une simple variante de l'intersection d'un rayon avec l'arbre de construction : $e_1>0 \cap e_2>0 \cap ... e_n>0$. Théoriquement, l'objet "polyèdre convexe" est inutile. Il est pourtant employé pour des raisons d'efficacité; en effet un volume englobant peut ainsi lui être attaché : il est inutile d'appeler la fonction intersection_avec_polyèdre quand le volume englobant n'intersecte pas le rayon.

3.2. Intersection avec une surface

3.2.1. Intersection avec une grille altimétrique

Le relief des terrains est le plus souvent décrit soit par un réseau de facettes, en général triangulaires (c'est le cas de beaucoup de simulateurs de vol), soit par une grille altimétrique régulière (i.e. en chaque point d'une maille rectangulaire on possède une altitude). L'intersection d'un rayon et d'un terrain modélisé par une grille altimétrique est maintenant étudiée.

La surface du terrain à l'intérieur d'une case de la grille peut être modélisée par deux triangles ou par une surface gauche. Par la suite cette description est appelée "facette". Pour déterminer l'intersection entre une grille altimétrique et un rayon, la méthode la plus simple consiste à tester toutes les facettes. Si la grille compte NxN cases, cette méthode est en $O(N^2)$ et est donc extrêmement inefficace. L'algorithme de Coquillart [COQU84] n'effectue que $O(N)$ tests d'intersection avec une facette :

Considérons la projection du rayon sur le plan de la grille. Il est évident que le rayon ne peut intersecter que les facettes des cases traversées par la projection du rayon; il suffit donc de tester ces facettes; les cases empruntées par le rayon en projection sont déterminées par une variante de l'algorithme de Bresenham. Cet algorithme peut être employé pour tous les types de rayons : primaires, secondaires, etc.

Une autre optimisation utilise l'absence de surplomb dans le terrain. Supposons, pour simplifier, que les verticales du monde restent verticales sur l'image; autrement dit le regard est horizontal, ou bien le regard est oblique mais l'œil est à l'infini. Considérons ensuite le plan vertical C, passant par le rayon primaire d'un pixel (i,j). Soit P le premier

point d'intersection entre le terrain et le rayon. Le rayon primaire pour le pixel (i,j+1) a la même projection que le rayon précédent, il appartient au plan C; son premier point d'intersection avec le terrain se trouve forcément "plus loin" que P par rapport à l'œil : il appartient soit à la même facette que P, soit à une facette encore plus loin de l'origine du rayon. D'où l'optimisation suivante : pour chaque colonne, les pixels sont calculés du bas vers le haut de l'image et les rayons primaires sont lancés à partir de la dernière facette intersectée. Quand le parallélisme des verticales n'est pas conservé dans l'image, ce type d'optimisation reste possible, comme le montre Coquillart; mais il est alors moins évident.

Formulé ainsi, cet algorithme est assez indépendant de la structure de données : un quadtree pourrait être employé plutôt qu'une simple grille régulière. Cette méthode peut aussi être appliquée quand le terrain est représenté par un réseau de facettes triangulaires : la facette contenant l'œil doit pouvoir être déterminée efficacement; de plus, pour pouvoir suivre rapidement la projection du rayon de facette en facette, il faut connaître pour chaque segment du réseau ses deux facettes.

Fondamentalement, cet algorithme exploite la "cohérence" des terrains, c'est à dire des surfaces du type z=f(x,y), comme l'ont fait avant lui d'autres algorithmes de parties cachées : Wright [WRIG73], Williamson [WILL72], Anderson [ANDE82].

3.2.2. Intersection avec une surface stochastique

Modifiant une idée de Mandelbrot [MAND77], Fournier et al [FOUR82] ont proposé un algorithme engendrant des surfaces stochastiques, ayant l'allure de terrains naturels. La surface stochastique est obtenue en divisant récursivement n fois un triangle initial en quatre autres triangles plus petits. A tous les niveaux de récursion, la division d'un triangle est effectuée ainsi :

- calculer le point milieu de chaque coté.
- ajouter à ces points un "déplacement" dans une direction préfixée. La longueur du déplacement est une fonction aléatoire gaussienne, dont la variance est proportionnelle au carré de la longueur du côté du triangle divisé.
- ces points, et les trois anciens sommets, définissent quatre nouveaux triangles. Appliquer les mêmes règles à chacun de ces triangles.

Le processus détermine ainsi les 4^n triangles "terminaux" de la surface stochastique. Pour calculer l'intersection entre une telle surface et un rayon, deux types de méthodes sont envisageables :

- engendrer et stocker les 4^n facettes terminales; calculer l'intersection de ces facettes et du rayon. L'algorithme déterminant l'intersection d'une grille altimétrique et d'un rayon reste bien sûr applicable.
- exploiter la définition récursive de la surface fractale, en n'engendrant, au vol,

que les triangles nécessaires. La description explicite de la surface, fort encombrante, n'est pas nécessaire, et le niveau de récursivité peut être arbitrairement grand. Cette méthode due à Kajiya [KAJI82] est maintenant détaillée.

Pour tous les niveaux de récursion, l'algorithme suppose qu'il n'est pas nécessaire de diviser le triangle pour connaître un volume englobant de la surface engendrée par ce triangle.

Bien sûr, les volumes englobants sont bornés. Aussi peut-il paraître abusif de parler de volume englobant, puisque le déplacement des milieux des segments est une fonction aléatoire gaussienne : un déplacement arbitrairement grand est donc possible, même s'il est très improbable. Deux attitudes sont alors possibles : soit on se contente d'un volume englobant probabiliste, qui a de très bonnes chances de contenir la surface; soit on utilise une fonction aléatoire à support borné : par exemple les valeurs de déplacement trop improbables sont tronquées.

La nature très particulière de la surface, permet de trouver un volume englobant seulement à partir de son triangle initial, par exemple un prisme à base triangulaire; l'axe du prisme est bien sûr la direction des déplacements des milieux des segments. Le volume englobant d'un triangle terminal est le triangle lui même.

L'intersection entre un rayon et une surface stochastique peut être déterminée par un algorithme non récursif; il gère une liste ordonnée L de triangles actifs :

1 La liste L contient seulement le triangle initial. Le volume englobant est censé intersecter le rayon.

2 Enlever de L le triangle de tête, t.

3 Si t est un triangle terminal (ou si son volume englobant est suffisamment petit), alors l'intersection est connue : vider L et sortir; sinon diviser t en 4 (Remarque : le déplacement du milieu d'un même segment peut être demandé plusieurs fois pour des rayons voisins; il va de soi que le déplacement rendu doit toujours être le même, pour engendrer toujours la même surface !)

4 Calculer l'intersection entre le rayon et les volumes englobant de nouveaux triangles. Rejeter les triangles sans intersection. S'il ne reste plus de triangles, alors aller en 2.

5 Trier les triangles restants dans l'ordre des distances croissantes à l'origine du rayon.

6 Insérer ces triangles, ordonnés, en tête de L. Aller en 2.

Le prisme triangulaire est le volume englobant le plus simple, mais il n'approche pas de très près la surface; les ellipsoïdes englobants se définissent de façon moins

évidente, mais sont préférables, comme le montre Bouville [BOUV85].

Des surfaces stochastiques peuvent aussi être engendrées par division récursive d'un carré en quatre carrés plus petits; un algorithme semblable dans le principe peut être employé.

3.2.3. Intersection avec une surface algébrique

3.2.3.1. Surface implicite f(x,y,z) = 0

Soit une surface algébrique donnée sous forme implicite : f(x,y,z)=0. Dans cette expression, x, y et z sont remplacées par leur valeur (x y z 1) = E + t $\vec{V}$. Une équation algébrique univariée en t est obtenue, et résolue par les méthodes numériques classiques.

Les premières méthodes envisageables sont les variantes de l'algorithme de Newton. Dans un article paru en 82 [KAJY82], Kajiya conseille la méthode de Laguerre, plus stable et de convergence plus rapide que la méthode de Newton; elle utilise la parabole tangente au lieu de la droite tangente ([RALS65]). Toujours d'après Kajiya, une autre méthode, d'un principe tout différent, est peut être intéressante; elle utilise les théorèmes bornant a priori les zéros réels des polynômes, et le théorème de Sturm; ce dernier permet de calculer le nombre de zéros, dans un intervalle réel quelconque, pour un polynôme donné (à cœfficients réels, et sans racines multiples). Une simple recherche dichotomique permet ensuite d'isoler les solutions réelles dans un intervalle aussi étroit que souhaité.

3.2.3.2. Surface paramétrée

La synthèse d'images utilise assez souvent des surfaces gauches paramétrées, définies sous la forme :

$$\{ F = x-f(u,v) = 0; G = y-g(u,v) = 0; H = z-h(u,v) = 0 \} \quad (3).$$

Les plus courantes sont les surfaces bicubiques, d'équation

$$(x\ y\ z\ 1) = (u^3\ u^2\ u^1\ u^0)\ P\ {}^t(v^3\ v^2\ v^1\ v^0) \quad (4),$$

où P est un tableau 4x4x4 de réels.

Seule une partie de ces surfaces paramétrées est utilisée, correspondant par exemple à l'intervalle $\{ 0 \le u \le 1, 0 \le v \le 1 \}$; d'où le nom de "carreau" de surface ("patch"); on parle aussi de "nappe".

Comment calculer l'intersection entre les surfaces algébriques paramétrées et un rayon ? Pour plus de commodité, l'exposé regroupe en trois classes distinctes les méthodes de calcul de l'intersection entre un rayon et une surface paramétrée. La première classe de méthodes emploie le principe informatique "diviser pour régner" en découpant récursivement la surface. Les deux autres classes font appel à des techniques numériques de la géométrie algébrique; la deuxième ramène le problème à la résolution numérique

d'une équation algébrique univariée; la troisième résout directement un système d'équations algébriques.

3.2.3.2.1. Subdivision récursive des carreaux de surface bicubiques

La première classe de méthodes est surtout applicable aux nappes bicubiques; elle utilise un découpage récursif de la surface. L'intersection entre le rayon et une nappe peut être réglée facilement dans les quatre cas suivants :

1. le volume englobant de la nappe n'intersecte pas le rayon;
2. le volume englobant intersecte le rayon, et il est "assez petit" vu de l'origine du rayon, pour qu'il soit inutile de découper davantage;
3. le volume englobant intersecte le rayon et il est "assez plat" pour qu'il soit inutile de continuer le découpage ;
4. le volume englobant intersecte le rayon, mais il est situé derrière une autre intersection précédemment trouvée avec la nappe.

Dans tous les autres cas, la nappe est découpée.

Cet algorithme suppose qu'il est possible de connaître rapidement une bon volume englobant d'une nappe bicubique, à tous les niveaux de récursion. A titre d'exemple, les surfaces de Bézier sont toujours englobées par l'enveloppe convexe de leurs points de contrôle; or toute surface bicubique a des "points de Bézier", qui peuvent être déterminés. Dans son principe, cette première méthode est tout à fait comparable à l'affichage ligne à ligne ("scan-line") des patches proposé par Lane et al [LANE79], et à la technique de subdivision de Catmull [CATM74].

3.2.3.2.2. Réduction à une équation univariée

La deuxième classe de méthodes réduit le problème à la résolution d'une équation algébrique univariée. Voici d'abord la méthode naïve, conceptuellement la plus simple, applicable (théoriquement...) à des surfaces algébriques paramétrées de degré quelconque. La surface est donnée sous la forme paramétrée :

$$\{ F = x-f(u,v) = 0; G = y-g(u,v) = 0; H = z-h(u,v) = 0 \}.$$

La méthode élimine u et v de ce système par trois calculs de résultants, et obtient la forme implicite de la surface S(x,y,z)=0 : un paragraphe précédent a traité l'intersection d'un rayon et d'une telle surface. F et G sont considérés comme des polynômes en u, et leur résultant $R_u[F,G]$ est déterminé : c'est un polynôme en v, x et y; F et H sont considérés comme des polynômes en u, et leur résultant $R_u[F,H]$ est déterminé : c'est un polynôme en v, x et z; ces deux résultants sont ensuite considérés comme des polynômes en v, et leur résultant $R_v[R_u[F,G], R_u[F,H]]$ est déterminé : c'est un polynôme S en x, y et z et S(x,y,z)=0 est la forme implicite de la surface. Quand le degré de S est élevé (108

pour une bicubique), la méthode naïve perd de son intérêt.

Kajiya [KAJI82] utilise des techniques similaires, mais d'une façon bien plus efficace que la méthode naïve; la droite du rayon, vue comme l'intersection de deux plans, a pour équation :

$$(a_i\ b_i\ c_i\ d_i)\begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = 0 \quad \text{(i variant de 0 à 1) (5).}$$

En substituant (4) dans (5), un système de deux équations bicubiques en u et v :

$$\{\ A(u,v) = a_{3,3}u^3v^3 + \ldots = 0;\ B(u,v) = b_{3,3}u^3v^3 + \ldots = 0\ \}\ \text{est obtenu.}$$

Il peut être interprété comme l'intersection de deux courbes algébriques dans le plan (u,v). u est solution de $R_v[A,B] = r(u) = 0$, qui est une équation algébrique univariée en u, de degré 18. Après résolution numérique de r(u)=0, u est remplacé par ses valeurs réelles u_k, zéros de r, dans :

$$\text{pgcd}(A(u_k,v), B(u_k,v)) = 0;$$

Les équations en v obtenues sont de degré 3 au maximum, et v, x, y, z se trouvent donc ensuite sans aucune difficulté. Cette méthode se généralise sans mal aux surfaces paramétrées plus complexes que les bicubiques; les degrés sont bien sûr modifiés.

Une autre idée, assez proche de la méthode naïve, mais bien plus efficace est la suivante : la description de la surface paramétrée par une forme implicite est utilisable dans le cas des "patches de Steiner"; en effet Sederberg [SEDE84] a démontré que la forme implicite de ces surfaces est de degré total 4. Aussi propose-t-il de modéliser les surfaces gauches par des "patches de Steiner". Il expose aussi une méthode spécifique pour obtenir la forme implicite de la surface, plus adaptée que les calculs de résultants. L'intersection entre le rayon et la surface se réduit ainsi à la résolution d'une équation univariée de degré 4.

3.2.3.2.3. Résolution numérique d'un système algébrique

La troisième classe de méthodes résout directement le système d'équations algébriques : $F(X) = \vec{0}$, généralement en faisant appel à l'itération de Newton à plusieurs variables; on le sait, le choix d'un bon point de départ est la principale difficulté de cette méthode; pour la surmonter, Toth [TOTH85] propose une analyse par intervalles, ou régions; une région est une boîte dans $\Re^3$, dont les faces sont parallèles aux axes; l'espace occupé par la surface est découpé en régions plus petites, que l'on sait traiter dans les cas suivants :

1. la région n'intersecte pas le rayon;
2. la région vérifie certaines contraintes (Toth propose deux tests, exploitant notamment le théorème de Krawczyk-Moore) assurant que la méthode de Newton converge bien;
3. la région est presque plate;
4. la région est trop petite pour être encore découpée en deux : elle est solution si elle est stable par une itération de Newton.

Les régions qui ne peuvent être traitées sont découpées en deux; les régions les plus proches de l'extrémité du rayon sont étudiées en premier; le point d'intersection le plus proche de l'extrémité du rayon est trouvé en premier, et l'algorithme peut alors être arrêté (sauf cas d'opération ensembliste). Quelques précautions sont nécessaires pour les rayons secondaires, car l'origine du rayon peut être justement la première solution.

Dans un contexte différent du tracé de rayons, Moore [MOOR77] a proposé d'autres méthodes déterminant des régions où la convergence des algorithmes itératifs est assurée. Les techniques mathématiques nécessaires (algorithme de Newton, résultant, opérateur de Krawczyk, etc) ne sont pas présentées ici.

Remarque : Les algorithmes déterminant l'intersection avec une surface stochastique, l'intersection avec une nappe bicubique avec subdivision récursive, et ce tout dernier algorithme ont un air de famille évident; ainsi, ils gèrent tous une liste active de sous problèmes, les problèmes trop complexes étant récursivement réduits à des problèmes plus simples.

3.2.4. Intersection avec une surface de révolution

Beaucoup d'objets se décrivent simplement grâce à des surfaces de révolution : verres, vases, bouteilles et récipients divers, colonnes, stylos et châteaux d'eau, pièces mécaniques ... Décrire cette surface comme une entité (axe, génératrice) est bien plus naturel que la décrire par un assemblage de patches. Mais le calcul de l'intersection nécessite de nouvelles méthodes.

3.2.4.1. La génératrice est une courbe algébrique implicite

Si la surface de révolution a pour axe Oz et pour génératrice une courbe algébrique $f(x,z)=0$, alors $R_t[\ f(t,z),\ t^2-(x^2+y^2)\ ] = F(x,y,z) = 0$ est l'équation sous forme implicite de cette surface. L'équation du tore peut être obtenue ainsi. Par application du théorème de Bezout, le degré de F est seulement le double de celui de f. L'intersection d'un rayon et d'une telle surface de révolution est donc ramenée au problème de l'intersection avec une surface algébrique sous forme implicite.

3.2.4.2. La génératrice est une courbe algébrique paramétrée

La surface de révolution a pour axe Oz et pour génératrice une courbe algébrique paramétrée z=g(u), r=r(u). Les points du rayon, transformés dans ce repère, sont de la forme (x y z)= E+t $\vec{V}$; les points d'intersection avec la surface vérifient de plus :

$$x^2+y^2 = r(u)^2,\ z = g(u), \text{ ou encore}$$

$$(E_x+tV_x)^2 + (E_y+tV_y)^2 = r(u)^2 \text{ et } E_z+tV_z = g(u).$$

Cette toute dernière expression permet d'exprimer t en fonction de g(u); une équation du type $Ag(u)^2 + Bg(u) + C = r(u)^2$ est obtenue; c'est une équation en u, de degré 6 dans le cas où g(u) et r(u) sont des polynômes cubiques. Sa résolution numérique donne les valeurs de u, puis de t, x, y, et z.

Bouville et al [BOUV84] utilisent pour les fonctions g(u) et r(u) des cubiques, modélisées par des B-splines; pour le calcul d'intersection, ces cubiques sont pourtant considérées comme des courbes de Bézier : ainsi une meilleure surface englobante peut leur être affectée.

3.2.4.3. La génératrice est une ligne brisée

Dans le cas où la génératrice n'est pas une courbe analytique, Kajiya [KAJI83] a proposé la méthode suivante. La génératrice g est initialement donnée par une succession de segments, c'est à dire n points (r_i,z_i), dans un repère de centre A et d'axes Ar, Az (r pour rayon); l'axe de la surface de révolution S est l'axe Az. Par commodité, l'appartenance d'un point (r,z) à la génératrice est notée g(r,z)=0, bien que l'expression analytique ne soit pas disponible.

Comme pour le prisme, l'intersection entre S et le rayon est d'abord étudiée dans un plan auxiliaire. Soit P le plan contenant le rayon et parallèle à l'axe Az de la surface S. Soit d la distance entre P et l'axe Az. Le plan est muni du repère orthonormé direct Oxyz, où O est la projection de A sur P, Oz est parallèle à Az, et Oy est sur la demi-droite OA; ceci fixe Ox. Ainsi, P est le plan y=0, et A a pour coordonnées (0 d 0) dans Oxyz. Le rayon est défini dans ce repère par : $x= x_0+ at$, $y= 0$, $z= z_0 + ct$.

L'intersection entre P et S est une courbe de points (x 0 z) vérifiant :

$$x^2 = r^2 - d^2 \text{ et } g(r,z)=0.$$

L'intersection entre le rayon et cette courbe est la solution; cette courbe peut être décrite par les deux successions de n points $(x_i\ z_i) = (\pm\sqrt{r_i^2-d^2};\ z_i)$. Hélas, comme cette expression dépend de d, il faudrait l'engendrer pour chaque rayon; quand n croît, ce calcul devient vite prohibitif; il est évité par le changement de variables : $X=x^2$, $R=r^2$.
Dans le repère ARz, la courbe génératrice $g(\sqrt{R},z) = 0 = G(R,z)$ est engendrée par la

succession des n points $(R_i, z_i) = (r_i^2, z_i)$. Ce calcul, indépendant du rayon, est effectué une seule fois.

Par ailleurs, $x^2 = r^2-d^2$ et $x = x_0+at$ impliquent $R = (x_0 + at)^2 + d^2$.

Dans l'espace (R,z), l'intersection cherchée est donc l'intersection entre la courbe G(R,z)=0 et un rayon devenu parabolique, qui est l'ensemble des points (R,z) tels que :

$$R = (x_0+at)^2 + d^2, \quad z = z_0 + ct.$$

L'intersection entre la courbe G et ce rayon se réduit à des intersections entre des segments et une parabole, c'est à dire à des équations du deuxième degré en t. La difficulté n'est pas de nature mathématique, mais informatique : n peut être grand, et effectuer n tests ne sera guère efficace. D'où l'idée de représenter la succession des (R_i, z_i) par un strip- tree.

Récapitulons l'algorithme. Un prétraitement commun à tous les rayons transforme la génératrice $g=(r_i,z_i)$ dans le repère ARz, et représente la courbe obtenue, $G=(r_i^2;z_i)$, par un strip-tree pour plus d'efficacité. Le rayon est transformé dans le repère ARz le ou les points d'intersection entre G et le rayon devenu parabolique :

$$R = (x_0+at)^2 + d^2, \; z= z_0 + ct$$

sont déterminés.

Connaissant un point d'intersection de paramètre t, il est facile d'en déduire les R, r, z, et x correspondants, la normale pour G puis la normale pour g. Une rotation de cette dernière donne ensuite la normale à S en (x y z). Les résultats connus dans Oxyz sont finalement replacés dans le repère du monde.

Les feuilles du strip-tree peuvent être des arcs de courbe plutôt que des segments; mais il est nécessaire de savoir calculer les intersections de ces arcs avec une parabole, et leurs rectangles englobants.

3.3. Intersection avec un octree

L'intersection d'un rayon avec un octree se réduit au problème suivant : trouver les feuilles de l'octree successivement traversées par le rayon à partir de son origine; le trajet du rayon dans l'octree n'est suivi que jusqu'à la première intersection. Le calcul de l'intersection entre le rayon et un voxel non vide dépend bien sûr de la définition d'un voxel (pointeur sur un objet, pointeur sur une liste d'objets, ou seulement un booléen : vide/plein), et ne nous intéresse pas ici.

Le voxel (feuille de l'octree) contenant l'origine du rayon est d'abord déterminé, ce qui ne pose pas de problème particulier ([GLAS84]). S'il n'y a pas intersection, comment passer au voxel suivant ? L'idée ([KUNI85]) consiste à calculer d'abord un point, N, qui se trouve forcément dans le voxel voisin; ensuite il ne reste plus qu'à trouver le voxel

contenant le point N. Pour calculer un point N correct, les coordonnées du point S où le rayon sort du voxel actuel sont déterminées; supposons que S se trouve sur la face verticale nord du voxel; à partir de S, un déplacement horizontal de epsilon en direction du nord donne un point N correct; en pratique, epsilon peut être la moitié (ou une fraction quelconque) du coté du plus petit voxel de l'octree. Sur ce principe, l'algorithme calcule le déplacement nécessaire dans toutes les directions pour passer de S à N : ceci permet de tenir compte des cas particuliers, où S se trouve juste sur une des arêtes ou un des sommets du voxel. D'autres méthodes de passage d'une cellule à la suivante ont été proposées : [FUJI83], [AMAN87].

Pour éviter les imprécisions numériques inhérentes à la représentation flottante, tous les calculs de suivi du rayon dans l'octree sont effectués en entier; c'est possible, car l'extrémité et la direction du rayon sont recalées sur des entiers, et les points d'intersection entre le rayon et les limites d'un voxel sont rationnelles; il va de soi que la géométrie de l'octree (coordonnées des sommets des voxels) n'utilise que des entiers. Par ailleurs, les calculs entiers sont plus rapides que les calculs flottants (sauf emploi d'un processeur spécialisé); il est bien sûr vital de ne pas perdre de temps (ou le moins possible) à traverser les voxels vides : les logiciels de tracé de rayons utilisent assez souvent la partition de la scène par un octree pour diminuer le nombre des tests d'intersection. Dans le principe, l'intersection d'un rayon avec un bin tree se calcule de la même façon.

3.4. Intersection avec divers objets

Nous nous contenterons de citer des références sur les calculs d'intersection avec les objets suivants :

- objets déformés : [BARR84], [BARR86].
- cylindres généralisés : [BRON85].
- particals : [INAK88].
- densités volumiques : [KAJI84].
- objets définis par balayage de splines cubiques : [WIJK84].

4. Optimisation des calculs d'intersection

Selon les critères actuels, le tracé de rayons nécessite énormément de calculs. Les calculs d'intersection entre les rayons et les objets de la scène sont les plus coûteux. Diverses optimisations logicielles ont été proposées depuis les débuts du tracé de rayons.

4.1. Dans l'espace objet

4.1.1. Volumes englobants

Le calcul d'intersection entre un rayon et un objet complexe (prismes, nappes bicubiques, etc) est très pénalisant en temps de calcul; or souvent, l'objet n'intersecte

même pas le rayon ! D'où l'idée d'utiliser un test simple, qui puisse écarter rapidement les objets complexes qui ne peuvent manifestement pas intersecter le rayon : un volume englobant est attaché à chaque objet complexe; quand le volume englobant n'intersecte pas le rayon, il en va de même de son objet. Le volume englobant est un compromis entre deux exigences : il doit approximer son objet au mieux, et son intersection avec le rayon doit se calculer le plus vite possible. L'intersection avec la sphère est sans doute celle qui se calcule le plus rapidement; aussi la sphère est elle très souvent utilisée comme volume englobant. Malheureusement, elle approche souvent fort mal les objets (Une sphère n'a évidemment pas besoin de volume englobant).

D'autres volumes englobants souvent employés sont les boîtes (des parallélépipèdes rectangles), dont les faces sont parallèles aux axes du monde. Il semble que l'intersection soit moins rapide que pour la sphère, mais que l'approximation soit meilleure : ceci compense cela.

Plutôt que d'utiliser des sphères englobantes, on peut envisager l'emploi d'ellipsoïdes englobants. L'approximation est meilleure, mais le calcul d'intersection moins rapide; de plus, dans le cas d'une structuration hiérarchique des objets dans la scène, la création de l'ellipsoïde englobant n'est pas évidente : quel est le meilleur ellipsoïde englobant deux ellipsoïdes donnés ?

Enfin, Kay et al. [KAY 86] proposent d'utiliser comme volume englobant des polyèdres dont les faces ont des normales particulières : parallèles aux axes du monde ou à 45 degrés avec les trois plans Oxy, Oxz et Oyz : ces particularités permettent d'optimiser les calculs d'intersection; l'approximation est bonne; en outre, il est assez facile de trouver le volume englobant plusieurs volumes englobants d'objets proches.

L'utilisation d'une structuration hiérarchique de la scène (arbre de construction par exemple) [RUBI80] permet en effet de regrouper des objets proches dans l'espace, et de leur attacher un volume englobant, d'une taille encore assez réduite. Si le rayon n'intersecte pas ce volume englobant, alors il est inutile de tester tous les objets du sous arbre : beaucoup de temps CPU est ainsi économisé. Il est évident que l'optimisation des volumes englobants, sans utilisation de la hiérarchie, ne diminue pas le nombre de tests d'intersection, mais ne fait que les accélérer. Il est bien difficile de donner une mesure théorique de l'efficacité de la hiérarchie, que confirment pourtant les mesures empiriques.

Comment calculer le volume englobant au nœud d'un arbre de construction binaire ? Soient u et v les deux objets fils, et U et V leurs volumes englobants respectifs. Le volume englobant l'union de u et v doit englober U et V; il est facile de trouver un tel volume dans le cas des boîtes et des sphères (considérer la droite joignant les centres de U et V et les points d'intersection les plus éloignés); un tel volume englobant n'est intéressant que si les deux objets sont assez proches. Le volume englobant l'intersection de u et v doit englober l'intersection de U et V; dans le cas des boîtes, l'intersection des boîtes U et V est elle même une boîte qui se trouve facilement; dans le cas des sphères, l'intersection de U et V est un cercle C : la sphère de grand cercle C est un volume

englobant correct pour u ∩ v. Le volume englobant la différence u-v est U.

4.1.2. Projection sur un plan

Une autre optimisation considère la projection des objets ou de leurs volumes englobants sur un plan de référence. Dans les cas les plus simples, ce plan est partitionné par une grille régulière (ou un quad tree); un prétraitement détermine pour chaque case de la grille l'ensemble des objets qui s'y projettent. La suite de cases dans lesquelles se projette le rayon est déterminée par une variante de l'algorithme de Bresenham; seuls sont testés les objets se projetant dans les cases où passe la projection du rayon (dans le cas le plus simple, le rayon est perpendiculaire au plan de projection, et une seule case est visitée). Comme le rayon est suivi de case en case, les points d'intersection sont à peu près trouvés dans l'ordre : du plus proche au plus éloigné de l'origine du rayon; or le premier point d'intersection est très souvent suffisant, ce qui permet une nouvelle optimisation. Sur un principe similaire, une autre partition du plan de référence, moins simple, a été proposée par Coquillart [COQU85] : la partition est décrite par une BREP. Dans tous les cas, il faut pouvoir "suivre" rapidement le trajet des rayons dans la partition du plan de projection. Ces techniques perdent bien sûr leur intérêt quand beaucoup d'objets se projettent au même endroit; l'emploi de deux plans de projection au lieu d'un est une solution.

Cas particulier : le plan de projection est le plan de l'image ([ROTH82], [WEGH84], [BRON84], [GERV86], [EXCO87], [VERR87] et [ARVO87]). La grille dans le plan peut alors être la matrice de pixels elle-même, ou un sous-échantillonnage de cette matrice.

Fondamentalement, ces techniques partitionnent l'espace en cellules, une cellule étant l'ensemble des points de $\Re^3$ qui se projettent dans une même case du plan de référence. Ces techniques exploitent donc l'idée de la partition spatiale :

4.1.3. Partition spatiale

Le principe de la partition spatiale est très souvent employé pour optimiser le tracé d'un rayon. L'espace est partitionné en cellules, de façon à vérifier les conditions suivantes : les objets (ou les volumes englobant des objets) occupant chaque cellule sont déterminés par un prétraitement, de préférence rapide; une cellule doit contenir un nombre restreint d'objets, si cela est possible; enfin, le trajet d'un rayon quelconque dans la partition spatiale doit pouvoir être suivi très rapidement. En conséquence, seuls sont testés les objets présents dans les cellules traversées; les points d'intersection sont trouvés quasi ordonnés : il suffit d'ordonner les points d'intersection trouvés dans une même cellule. Très souvent le premier point d'intersection est suffisant; d'où une autre optimisation possible, évidente.

La partition spatiale peut être réalisée par une grille bidimensionnelle structurée en quadtree, mais pourrait l'être aussi par un tableau de voxels ou grid , un octree, ou un bin

tree. Nous détaillerons d'abord l'utilisation d'octree puis nous présenterons une méthode de cohérence spatiale.

4.1.4. Octree

4.1.4.1. Octree de Glasner

La première optimisation du tracé de rayons par emploi d'octree ([GLAS84]) attache à chaque feuille de l'arbre, la liste des objets dont le contour intersecte le voxel ; plus précisément cette liste est une liste de pointeurs sur des objets, qui ne sont décrits qu'une fois, hors de l'octree. Cette liste peut bien sûr être vide. Elle a une taille maximale préfixée : T. Cet octree ne sert qu'à connaître les surfaces présentes dans les voxels : la notion de voxel plein ou de voxel vide n'est pas utilisée. Dans le cas où la scène est décrite par un arbre de construction, un voxel peut donc fort bien pointer sur une surface qui n'existe pas dans la réalité, une fois effectuées les opérations ensemblistes ; ceci peut paraître un peu maladroit, cependant les mesures empiriques démontrent que l'octree garde une taille limitée et permet des gains en temps de calcul considérables, relativement à la version naïve du tracé de rayons.

Comme les opérations ensemblistes sont effectuées par le tracé de rayons, et non par l'octree, il n'est pas toujours possible de s'arrêter au voxel du premier point d'intersection.

Bien que ce soit improbable, cette méthode peut poser une difficulté quand plus de T surfaces d'objets passent au même endroit ; dans un autre contexte, Mantyla [MANT83] est confronté au même problème, et expose une solution.

4.1.4.2. Octree de Kunii et Wyvill

La deuxième optimisation [KUNI85] utilise l'octree pour, en plus, effectuer les opérations ensembliste : la scène est initialement décrite par un arbre de construction. Dans cette version, un voxel (feuille de l'octree) est soit vide, soit plein, soit "rebut", soit "de contour".

La définition d'un voxel vide ou plein va de soi ; quand un voxel est plein, l'objet qui l'englobe n'a pas besoin d'être mémorisé. Un voxel "de contour" est traversé par le contour d'un seul objet, et il pointe vers la description de cet objet ; il faut aussi mémoriser si cet objet est ajouté ou soustrait à la scène ; un voxel de contour stocke donc une sorte de minuscule arbre de construction, ne contenant qu'un opérateur ensembliste et qu'un seul objet (le lecteur comprendra ultérieurement la nécessité de cette remarque). Un voxel non plein, non vide, et non de contour, est "rebut" : un voxel rebut est trop petit pour être encore divisé ; la profondeur maximale de l'octree est en effet fixée a priori, selon l'application et le degré de précision souhaité (et la place mémoire disponible !).

L'octree représentant un objet non composite se réduit à un unique voxel de contour, pointant sur cet objet. L'octree représentant toute la scène se construit ensuite par

des opérations ensemblistes successives entre les octrees représentant les objets non composites. Toutes les opérations ensemblistes s'expriment en fonction du complémentaire et de l'intersection ; la première est triviale ; la deuxième s'effectue ainsi :

4.1.4.3. Fusion de deux octrees

L'intersection entre deux nœuds s'effectue récursivement sur leurs fils ; l'intersection entre un voxel vide et un autre octree est le voxel vide ; l'intersection entre un voxel plein et un autre octree est l'autre octree ; l'intersection entre un voxel rebut et un nœud ne peut se produire ; l'intersection entre un voxel rebut et un voxel ni vide et ni plein est le voxel rebut ; tous les autres cas se règlent par division récursive des voxels, par exemple l'intersection entre deux voxels de contour.

Il n'est donc jamais nécessaire de savoir calculer explicitement l'intersection entre deux surfaces, contours d'objets ; encore faut-il savoir diviser un voxel de contour : soit un voxel intersecté par le contour d'un objet ; quels fils de ce voxel sont intersectés par le contour de l'objet, lesquels sont hors de l'objet, et lesquels sont contenus dans l'objet ? Pour l'instant, les articles ne traitent que les contours plans et les quadriques. Il n'est pas difficile de proposer une méthode un peu plus générale, applicable aux objets dont les points intérieurs vérifient une inéquation algébrique $f(x,y,z)<0$.

4.1.4.4. Intersection d'une surface et d'un voxel

La surface de contour intersecte un voxel si elle intersecte au moins l'une des faces du cube correspondant au voxel, ou si l'objet est inclus dans le voxel..., ou si une partie connexe de l'objet est incluse dans le voxel : l'objet n'est pas forcément connexe.

Pour l'exemple, la suite considère une face verticale :

$$z=z_0,\ x_0 < x < x_1,\ y_0 < y < y_1.$$

La surface intersecte ce carré si la courbe algébrique $f(x,y,z_0)=F(x,y)=0$ intersecte une des arêtes, ou si le carré contient une composante connexe de la courbe F.

L'intersection entre une arête et la courbe F est facile à trouver ; par exemple, l'intersection avec $y=y_1$ se réduit à la résolution de $f(x,y_1,z_0)=e(x)=0$, une équation univariée de degré inférieur ou égal à celui de f.

Le carré contient une partie connexe de F si et seulement s'il contient un point de F à tangente horizontale, ou un point multiple, qui vérifie : $F(x,y)=F_x'(x,y)=0$. Autre méthode possible, quand la connexité de F est certaine (par exemple si l'objet est convexe), et qu'un point P de F est connu : F est incluse dans le carré si et seulement si P y est inclus.

Si le contour de l'objet n'intersecte aucune des faces du voxel, il se peut que le voxel contienne une composante connexe de l'objet, et soit donc malgré tout un voxel de contour ; mais alors ce voxel contient un point du contour de l'objet à normale verticale,

ou bien un point multiple, qui vérifie le système $f(x,y,z)=f_x'(x,y,z)=f_y'(x,y,z)=0$. Certains raccourcis sont possibles en cas de connexité ou de convexité de l'objet.

Quand le contour de l'objet n'intersecte pas le voxel, Kunii et Wyvill doivent décider si ce voxel est à l'intérieur ou à l'extérieur de l'objet. Soit S un point quelconque du voxel, le signe de f(S) suffit pour indiquer l'intérieur de l'objet.

A notre connaissance, aucun article n'a encore proposé de méthode spécifique pour diviser les voxels de contour contenant des surfaces paramétrées.

4.1.4.5. Comparaison des octrees

Kunii et Wyvill coupent l'espace jusqu'à séparer les surfaces, ou jusqu'à atteindre les voxels rebuts ; tous les points des courbes d'intersection entre deux surfaces appartiennent donc à des voxels rebuts ; cet octree est beaucoup plus encombrant que celui de Glasner et les scènes traitées sont d'ailleurs moins complexes. Il semble qu'énormément de place mémoire serait économisée si les voxels de contour pouvaient contenir un minuscule arbre CSG de un ou deux objets. Plus généralement, un voxel de contour pourrait contenir un mini-arbre de construction d'une taille maximale préalablement fixée : les procédures de fusion de deux octree restent inchangées dans le principe, de même que l'utilisation de l'octree par le tracé de rayons.

4.1.5. Structures de cohérence spatiale

Arnaldi et al. ([ARNA87]) n'utilisent pas une subdivision régulière comme Glassner ou Kunii et Wyvill, mais ils travaillent sur une division irrégulière de l'espace. Lors d'une première phase, chaque primitive voit son volume englobant réduit par les opérations booléennes de l'arbre CSG (cf fig suivante).

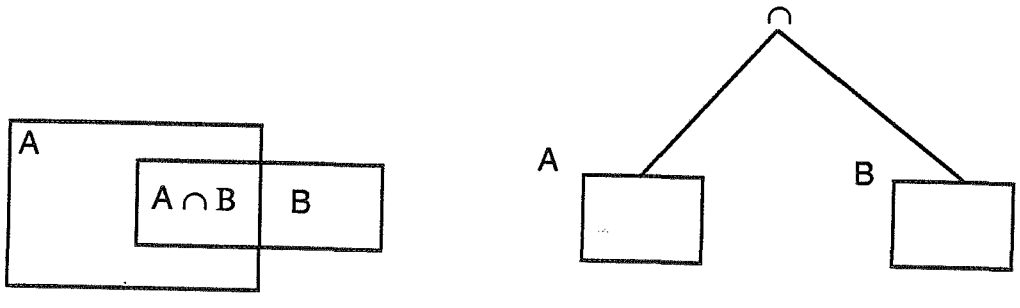


Fig 7. Réduction des boîtes englobantes

Le plan contenant l'écran est choisi comme plan de projection : en effet, chaque boîte englobante est projetée sur ce plan selon la position de l'œil. La projection d'une boîte sur l'écran permet d'obtenir un rectangle, celui-ci est découpé en quatre segments de droite. Cet ensemble de segments est ensuite utilisé pour réaliser une partition de l'écran

par la technique du BSO ([FUCH80]). Une décomposition de l'écran en rectangles est obtenue. Chacune des zones obtenues est ensuite découpée dans la troisième dimension par l'utilisation de valeurs englobantes sur cet axe. A chaque région 3D sont associés des pointeurs sur les régions voisines permettant le passage d'une région à la suivante pour suivre le trajet d'un rayon.

Dans [BOUA87] est proposé une autre méthode utilisant aussi la réduction des boîtes englobantes. Mais les boîtes englobantes ne sont plus projetés sur l'écran. L'espace de la scène est découpé à l'aide d'un bintree, cad l'espace est divisé selon les plans perpendiculaires aux axes du repère de l'œil. A chacune des cellules est associée une liste des objets contenus. Une structure de grid (ou voxels) est ensuite construite Chaque cellule est numérotée, pour y accéder, on utilise une table de référence ([TAMM82], [TAMM84]). Cette table est un tableau à trois dimensions dont chacune des cases contient le numéro de la cellule.

4.2. Dans l'espace écran

4.2.1. Sous-échantillonnage

Le sous-échantillonnage est l'optimisation la plus simple ; il peut d'ailleurs être employé de pair avec un sur-échantillonnage local pour pixels aliassés. On part d'un échantillonnage réduit de l'écran, on calcule les valeurs de ces pixels. Ensuite on déduit récursivement, la valeurs des pixels non calculés, soit par un mélange des valeurs des pixels voisins s'il n'y a pas de brusques différences entre ces voisins ; sinon on sur-échantillonne les zones non calculées. Cependant, des contours d'objets petits ou minces risquent fort d'être oubliés.

Une méthode classique ne calcule qu'un pixel sur deux (par exemple ceux dont l'abscisse et l'ordonnée ont des parités égales) ; la couleur des pixels non calculés est un mélange de la couleur des quatre pixels voisins.

4.2.2. Boîtes avec faces parallèles à l'écran

Roth [ROTH82] emploie des boîtes englobantes avec des faces parallèles aux bords de l'écran ; l'intersection entre le rayon et une boîte se réduit alors au test de l'inclusion d'un point dans un rectangle parallèle aux axes de l'image ; il est possible de gérer en plus une liste des rectangles actifs pour la ligne d'image calculée ; l'intersection entre le rayon et une boîte active se réduit alors à l'inclusion d'un nombre dans un intervalle. Cette méthode teste encore tous les intervalles actifs pour chaque pixel calculé ; il est possible de faire encore mieux : avant le calcul de la ligne courante, un prétraitement attache à chaque pixel la liste des intervalles actifs qui lui sont superposés.

Comme les boîtes ne sont pas toujours une bonne approximation de l'objet englobé, Bronsvoort et al [BRON84] utilisent comme volume englobant des intervalles correspondant à chaque ligne de l'écran.

Ces volumes englobants sont bien sûr moins efficaces pour les rayons non primaires ; de plus ils sont dépendants du point de vue : ils doivent être recalculés pour chaque image.

En fait, la meilleure optimisation pour les rayons primaires n'utilise pas le tracé de rayons, mais un autre algorithme.

4.3. Utilisation d'un autre algorithme pour les rayons primaires

Autre optimisation, les impacts des rayons primaires peuvent être déterminés par un algorithme de visualisation plus rapide ; ce dernier doit déterminer l'objet vu en chaque pixel, et bien sûr donner le même résultat que l'algorithme de la force brutale, ce qui exclut le recours à des facétisations naïves. La méthode idéale est sans doute une variante de l'algorithme par balayage de Watkins [WATK70] ; cette variante doit traiter les surfaces gauches, en utilisant par exemple la technique de Lane et al [LANE79] ; elle doit aussi traiter les objets composites, par la technique de Atherton [ATHE83]. Cette optimisation est sans doute la plus efficace qui puisse être proposée ; hélas, ces techniques élaborées sont d'une programmation assez difficile ; le tracé de rayons perd ainsi en simplicité ce qu'il gagne en efficacité.

Cette optimisation est sans effet sur les rayons non primaires. Des méthodes spécifiques permettent d'optimiser certains rayons secondaires.

4.4. Pour les sources lumineuses

Comment optimiser le tracé des rayons vers les sources lumineuses ponctuelles ? Les méthodes s'inspirent d'un principe classique, qui considère une source lumineuse comme un œil virtuel : cet œil voit ce qui est éclairé par la source. D'où l'idée de calculer des perspectives auxiliaires qui permettront de savoir si un point est vu/éclairé par un œil/source. Cette idée simple se heurte à quelques difficultés. Par exemple, une source peut émettre dans toutes les directions de l'espace, alors que le champ visuel est limité à la pyramide de vision ; plusieurs perspectives auxiliaires sont donc nécessaires pour une même source.

Haines et Greenberg [HAIN86] ont implanté une méthode de ce type, qui améliore les performances d'un facteur de quatre à trente. Un cube virtuel est centré sur chaque source ; chacune des faces du cube est partitionnée par une grille carrée basse-définition, de 40x40 cases par exemple. La faible définition s'explique surtout pour des raisons d'encombrement mémoire. Un pré-traitement détermine la liste ordonnée des objets dont les facettes de projettent sur chacune des cases. Si une case est entièrement couverte par une facette, il est inutile de garder la liste des objets qui suivent, cela réalise un gain de place mémoire.

Considérons maintenant l'impact I entre un rayon et un objet de la scène ; est-il ou

non éclairé par la source S ? I se projette sur une case C d'une des faces du cube virtuel de S. Cette case est déterminée. Soit LC la liste des objets se projetant sur C. I est éclairé par S si et seulement si aucun des objets de LC ne se trouvent entre I et S ; en général, cette liste est bien plus réduite que le nombre des objets de la scène, d'où l'optimisation.

Le lecteur aura reconnu là une adaptation de la technique qui projette sur l'écran les boîtes englobantes des objets. D'autres optimisations sont utilisées : supposons que le même objet, A, soit vu en deux pixels voisins, et que A soit dans l'ombre d'un objet B pour le premier pixel ; quand l'éclairement au deuxième pixel est calculé, et que l'algorithme teste les objets susceptibles de porter ombre sur A, il vaut mieux commencer par B : il est fort probable que B masque encore A pour le deuxième pixel.

5. Rendu et considérations optiques

Le principal intérêt du tracé de rayons vient de ce qu'il prend tout naturellement en compte la plupart des effets optiques. Nous allons donc exposer dans ce paragraphe le modèle d'éclairement généralement utilisé en tracé de rayons, les effets simples de bleu atmosphérique et de brouillard et la simulation de surfaces granuleuses par perturbation de la normale. Ensuite nous parlerons des effets que le tracé de rayons, dans sa forme la plus simple, ne parvient pas à prendre en compte, ce qui nous amènera à introduire une extension de l'algorithme du tracé de rayons que nous appellerons tracé de rayons distribué. Mais tout d'abord, nous présenterons une méthode d'affichage en fil de fer qui utilise la méthode du tracé de rayons.

5.1. Images de type fil de fer

En 1982, Roth [ROTH82] a proposé l'utilisation du tracé de rayons pour les visualisations interactives en CAO-CFAO. Les images obtenues sont du type fil de fer, avec effacement des parties cachées, sans aucun effet optique. La méthode peut être facilement programmée sur micro-ordinateur pour s'initier à l'infographie; elle rend une liste d'arêtes à afficher; les scènes doivent cependant rester assez simples.

L'idée consiste à calculer une image virtuelle basse définition (disons 128x128). Cette image est virtuelle car elle n'a pas besoin d'être affichée, ni même besoin d'être stockée en entier : une ligne suffit. En chaque pixel de cette image, l'objet vu est déterminé par l'algorithme du tracé de rayons; les rayons primaires sont bien sûr suffisants et les normales sont inutiles. Quand les objets vus (ou les faces vues, pour les objets polyédriques) diffèrent pour deux pixels consécutifs, une arête d'interpixel, horizontale ou verticale, a été détectée. Les premiers contours obtenus présentent un fort aliassage, mais Roth les affine ensuite en lançant des rayons primaires supplémentaires; partir d'une image virtuelle haute définition serait plus simple mais moins efficace. Hélas, si Roth montre des images fil de fer, il ne détaille guère son procédé.

On peut cependant imaginer la méthode suivante pour des visualisations fil de fer :

calculer l'objet vu en chaque sommet d'une grille-écran 128x128; les rayons ne passent donc pas par le centre des pixels, mais par les sommets. Si les quatre objets (ou faces) vus aux sommets d'un pixel diffèrent, alors le pixel contient un contour et il est coupé en quatre sous-pixels égaux; les objets vus aux nouveaux sommets sont déterminés en lançant des rayons primaires supplémentaires; l'intersection de ces rayons n'est pas testée avec toute la scène, mais seulement avec les objets vus aux quatre sommets du pixel père : optimisation simple mais particulièrement efficace. Chacun des sous pixels peut récursivement être étudié par la même méthode, jusqu'à une taille limite, par exemple 1024 sous-pixels terminaux par lignes. Les arêtes horizontales et verticales se déduisent ensuite trivialement, en joignant les sous-pixels terminaux contigus qui contiennent un contour.

Par cette méthode vraiment très simple, il est facile d'obtenir des images au trait d'objets composites, de fractals, et de surfaces gauches : le tracé de rayons est incontestablement la méthode la plus gratifiante ! Les optimisations exposées plus loin (volumes englobants, partitions spatiales, etc) sont applicables. Les seuls défauts obtenus sont dûs à l'oubli des très petits objets, ou des objets longs et très minces : on retrouve là un problème classique d'aliassage (voir le chapitre 3).

5.2. Rappel du modèle d'éclairement de Whitted

Sans rentrer dans le détail des modèles d'éclairement [PERO88], le modèle généralement utilisé en tracé de rayons, introduit par Whitted [WHIT80], est le suivant:

$$I = I_a + k_d \sum_{j=1}^{l} (\vec{N} . \vec{L_j}) + k_s S + k_t T.$$

Où I est l'intensité renvoyée

I_a est l'intensité due à la lumière ambiante,

k_d est le cœfficient de lumière diffuse,

$\vec{N}$ est le vecteur normal à la surface (unitaire),

$\vec{V}$ est le vecteur incident (unitaire),

L_j est le vecteur unitaire dirigé vers la source lumineuse j,

k_s est le cœfficient de réflexion spéculaire,

k_t est le cœfficient de réfraction,

S est l'intensité lumineuse incidente dans la direction $\vec{R}$ (réflexion),

T est l'intensité de lumière provenant de la direction $\vec{P}$ (réfraction).

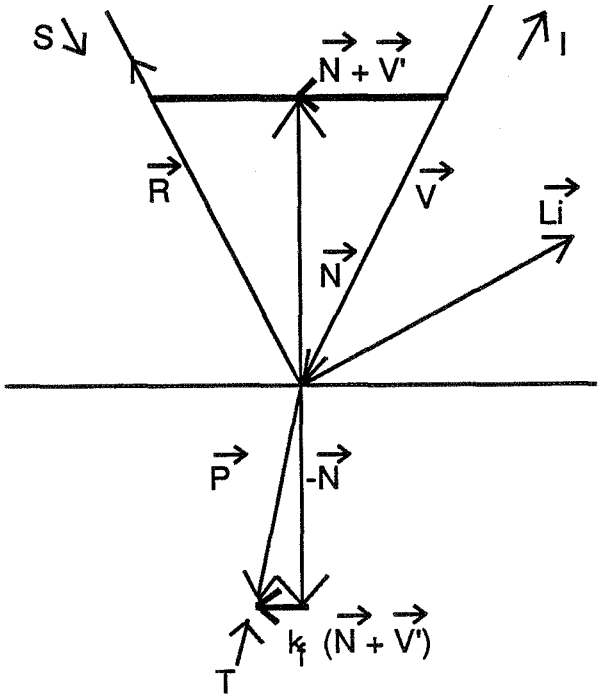


Fig.8 Surface intersectée.

Posons $\vec{V'} = \vec{V} / |\vec{V}.\vec{N}|$ alors $\vec{R} = \vec{V'} + 2\,\vec{N}$ et $\vec{P} = k_f(\vec{N} + \vec{V'}) - \vec{N}$

avec $k_f = \dfrac{1}{\sqrt{(k_n^2\,\|\vec{V'}\|^2 - \|\vec{V'}+\vec{N}\|^2)}}$, k_n étant l'indice de réfraction de la surface.

Depuis des modèles d'éclairement plus sophistiqués ont été développés par Hall et Greenberg [HALL83], Bouville et al. [BOUV84], Tezenas Du Montcel et Nicolas [TEZE85].

5.3. Bleu atmosphérique et brouillard

Comme le font les peintres paysagistes, on rajoute du bleu aux objets éloignés pour simuler l'épaisseur de l'air. Avec le tracé de rayons, la distance entre l'œil et l'objet vu est connue; la couleur de l'objet est modifiée par une quantité de bleu qui est une fonction exponentielle de la distance.

De même, le brouillard est simulé en ajoutant du blanc à la couleur des pixels, ce qui revient à ajouter la même quantité dans les composantes rouge, verte, bleue. Cette quantité est aussi une fonction exponentielle de la distance entre l'œil et l'objet vu.

5.4. Tracé de rayons distribué

Cette méthode permet de simuler : les ombres douces, la pénombre et les sources non ponctuelles, l'effet de profondeur de champ, les miroirs non parfaits, la translucidité, l'antialiassage temporel. Tous ces phénomènes sont simulés en utilisant une même technique d'échantillonnage ([COOK84]) : le tracé de rayons distribué. Nous allons d'abord expliquer la provenance et la difficulté de réalisation de ces effets.

En tracé de rayons classique, l'objet intersecté par un rayon issu d'un pixel de l'image est à l'ombre si un autre objet fait écran, éclairé sinon. On obtient ainsi comme résultat des frontières très nettes entre les zones éclairées et les zones à l'ombre, ce qui nuit au réalisme de l'image. D'où la nécessité de créer des ombres douces.

Les sources utilisées dans un tracé de rayons classique sont ponctuelles. Donc , les effets de pénombre ne peuvent être pris en compte. Une solution irréalisable pour créer l'effet de sources non ponctuelles (néons, ampoules...) serait de relancer une infinité de rayons et non plus un seul vers cette source lumineuse.

Si on veut simuler la prise d'un cliché par un appareil photographique on se heurte au problème de flou des plans proches ou éloignés. En effet, quand une photo est prise, seuls les objets aux environs de la distance focale de l'objectif sont nets, ceux qui sont trop proche de l'appareil ou trop éloignés sont flous. Ce qui permet de mettre en valeur le sujet principal de la photographie. La cause de ce flou est la suivante : l'objectif peut être assimilé à une lentille et celle-ci transforme tout point de son plan focal en un point sur l'image. Mais les points n'appartenant pas à ce plan sont projetés dans un cercle (cercle de confusion) sur l'image. Par conséquent, l'intensité de chaque pixel est la moyenne de la contribution de tous les objets dont le cercle de confusion passe par ce pixel.

Un miroir parfait renvoie un rayon dans une seule direction. Mais si la surface n'est pas un miroir parfait, le calcul d'intensité doit normalement être fait en intégrant la lumière réfléchie dans un angle solide dont la taille est déterminée par la nature de la surface. Le cas d'un objet translucide (i.e. un objet admettant une réfraction diffuse et donc les objets que l'on voit à travers sont flous) est le même : il faut intégrer, dans un angle solide lié à la nature de la surface, la lumière transmise. Ces deux intégrales étant définies de façon continue il faudrait envoyer une infinité de rayons pour bien simuler ces effets optiques.

L'antialiassage temporel est nécessaire pour les objets qui se déplacent très rapidement compte tenu de la fréquence d'échantillonnage pour créer une séquence animée (environ 25 images par seconde) ; en effet si leur vitesse est trop élevée, ces objets scintilleront sur l'écran. Pour éviter ce problème il suffit de représenter la trajectoire de ces objets pendant $\frac{1}{50}$ ième de seconde. Mais ceci demande de calculer une infinité d'images successives de ces objets, puisque le temps est continu.

L'échantillonnage permet de reconstituer un phénomène continu en l'échantillonnant de façon suffisante pour pouvoir recréer le phénomène (cf théorème de SHANNON,

chapitre 3). Le tracé de rayon distribué consiste donc à sur-échantillonner de façon indépendante, non seulement l'espace de l'image mais aussi l'espace temps, l'espace de réfraction, l'espace de réflexion .... C'est donc un sur-échantillonnage dans un espace à n dimensions où n représente le nombre de paramètres de toutes les fonctions à sur-échantillonner (fonction temporelle , de réflexion , de réfraction, d'illumination ...).

Le principe consiste donc à relancer des rayons en suivant soit :

- la fonction de réflexion spéculaire pour simuler les surfaces brillantes qui ne sont pas des miroirs parfaits;
- la fonction de transmission pour des surfaces qui ne sont pas parfaitement transparentes;
- la fonction d'éclairement d'une source non ponctuelle;
- la loi optique d'une lentille pour simuler la profondeur de champ;
- le temps pour antialiasser temporellement.

A chaque rayon relancé est associé un poids en général inversement proportionnel à la distance de ce rayon avec le rayon "type", c'est à dire celui lancé dans un tracé de rayons classique. L'utilisation d'une fonction de perturbation aléatoire pour relancer les rayons permet d'éviter les problèmes de maillage trop régulier qui nécessitent de renvoyer plus de rayons pour avoir le même effet car sinon des artefacts liés à la régularité sont introduits ([LEE85], [DIPP85], [PURG86]).

Le tracé de rayons distribué est très coûteux en temps de calcul car il demande de lancer, par pixel, de 4 à 256 rayons et en moyenne entre 14 et 49, selon les auteurs. Mais les effets obtenus sont d'un réalisme stupéfiant.

5.5. Tracé de rayons et antialiassage

Sans exposer sur les techniques de postfiltrage [GHAZ85], nous allons présenter les techniques actuelles d'antialiassage du tracé de rayons. Puisque les phénomènes d'aliassage proviennent d'un échantillonnage insuffisant lors du processus de visualisation, une méthode possible pour résoudre ce problème est un sur échantillonnage global de l'image. Par exemple, chaque pixel peut être divisé en quatre sous-pixels, et il faut donc à peu près quatre fois plus de temps pour calculer l'image; en général, le nombre de pixels aliassés est très inférieur au nombre total de pixels de l'image, et un sur échantillonnage local est suffisant et bien plus rapide.

Pour reconnaître les pixels aliassés, Whitted [WHIT80] mesure les sauts d'intensité sur quatre pixels voisins. Quand ce saut dépasse un certain seuil, les quatre pixels sont sur-échantillonnés. Cette méthode a pour inconvénient de sur-échantillonner les zones texturées de l'image, alors que des techniques spécifiques d'antialiassage des textures

seraient préférables.

Roth [ROTH82] ne compare pas l'intensité des pixels voisins, mais les objets vus sur deux pixels consécutifs de la ligne calculée. Si l'objet vu en (i,j) diffère de celui vu en (i+1,j), un rayon est lancé juste au milieu des deux pixels.

Une autre technique, basée sur la représentation par arbre de construction, compare neuf pixels voisins et ne relance les rayons supplémentaires que sur la sous-scène des objets vraiment concernés : d'où une optimisation importante (cf chapitre 3).

6. Parallélisation

Par sa nature le tracé de rayons est un algorithme hautement parallélisable. En effet, le calcul d'un pixel est entièrement indépendant des autres pixels. Ce qui a amené une première parallélisation simple du tracé. Nous pouvons faire la même séparation espace-écran / espace-objet que celle faite dans le paragraphe consacré aux optimisations. En effet, un microprocesseur peut être affecté à une zone écran , ou à une zone de l'espace objet. Par contre nous ne présenterons pas la vectorisation du tracé de rayons proposée par Plunkett et al. [PLUN85], car le gain obtenu, environ trente pour cent, ne parait pas satisfaisant compte tenu du fait que ce gain est relatif à un algorithme qui calcule l'intersection des rayons avec toutes les surfaces de la scène, sans aucune optimisation.

6.1. Un microprocesseur par pixel ou presque

Pour paralléliser le tracé de rayons dans l'espace écran, deux types de solutions existent : la première met l'ensemble des microprocesseurs en concurrence, chacun piochant dans le tas des rayons à calculer. La seconde utilise la cohérence dans l'espace écran ce qui revient à affecter non seulement des pixels mais aussi des objets à chaque microprocesseurs. Les rayons secondaires dans le premier cas sont traités par le processeur qui les génère, dans le deuxième cas ils sont transmis au microprocesseur qui se situe dans la direction des rayons lumineux. Ce deuxième cas demande une gestion plus compliquée et qui ressemble plus à ce que l'on va détailler dans le paragraphe suivant.

Reprenons la première méthode ([BOUV84]) ; chaque processeur traite l'arbre de réflexion-réfraction-ombre hérité du pixel qu'il calcule. Ce qui amène une situation de conflit pour l'accès à l'affichage sur l'écran et pour l'accès à la base de données de la scène. Mais l'accès à l'affichage n'est utilisé qu'une fois par pixel. Il suffit donc de contrôler l'accès à l'affichage par un partage temporel. Par contre, l'accès à la base de données est plus problématique, les processeurs ayant besoin d'y accéder fréquemment. Deux solutions sont envisageables :

- chacun des processeurs garde dans sa mémoire la base de données au complet ;

- chaque processeur charge en mémoire uniquement une partie de la base de données.

Dans le premier cas, comme une telle architecture est prévue pour des processeurs très simples, leur mémoire n'est pas très grande ce qui limite la taille de la base de données. Cette méthode a été réalisée sur une machine développée par le CCETT : CRYSTAL. La deuxième méthode oblige à bien "paginer" la base de données de telle sorte que le chargement en mémoire locale soit le moins fréquent possible compte tenu des impératifs de taille maximum. En terme d'arbre de construction, cela demande de charger un sous-arbre de taille limitée. Ce n'est pas un problème simple car la pagination doit se faire dans un espace à trois dimensions.

6.2. Un microprocesseur par cellule

Dans l'absolu, un microprocesseur par objet serait l'idéal ; hélas, à l'heure actuelle, une telle performance technique n'est pas envisageable pour une scène composée de milliers d'objets. L'idée est donc de regrouper ces objets dans des unités de volumes et de redistribuer ces volumes aux processeurs. Plusieurs approches ont été étudiées : la première ([DIPP84]) utilise des régions limitées par six faces en forme de losanges (rhomboèdre non régulier) qui contiennent a priori un nombre quelconque d'objets . Si une région contient trop d'objets, elle est modifiée pour distribuer un certain nombre d'objets à ses voisins. L'avantage de cette forme de région est que cette modification n'est que locale, ce qui n'est pas le cas d'une partition régulière de l'espace par exemple. L'inconvénient majeur de cette redistribution de l'espace est qu'elle peut amener des oscillations autour de certaines distributions, ce qui doit être détecté. La deuxième méthode, développée par Cleary et al [CLEA86] , utilise un réseau carré ou cubique de processeurs. Dans le cas d'un carré, il se trouve dans l'espace écran ; dans le cas d'un cube, dans l'espace objet ; les surfaces contenues dans la scène sont distribuées aux processeurs (dans le cas du carré c'est simplement une projection sur le réseau ; dans le cas d'un cube cela revient à représenter la scène avec des voxels-processeurs).

Cette dernière méthode a l'avantage d'avoir la même simplicité conceptuelle que le tracé de rayons. Tant qu'il ne reçoit pas de message de ses voisins et qu'il n'a pas de calcul d'intersection à faire, chaque processeur P appartenant à la face du cube liée à l'écran (i.e. chaque processeur dans le cas d'un carré), lance un nouveau rayon, et, calcule l'intersection de ce nouveau rayon avec les surfaces qu'il possède dans sa mémoire locale. S'il n'y a pas intersection, il calcule le point de sortie du volume élémentaire qu'il contrôle et détermine donc le prochain processeur P' concerné. Il envoie le paquet d'informations décrivant le rayon à P'. Si au contraire il y a intersection, alors selon la nature de la surface, P va recréer des rayons secondaires qu'il va traiter de la même manière qu'un rayon primaire.

Si le processeur P reçoit un message, celui-ci peut lui demander :

- de calculer des intersections entre le rayon transmis et les surfaces qu'il

possède ;

- de transmettre à un autre processeur le message ;
- de calculer une intensité si le message était un résultat d'intersection avec un rayon primaire i ou un de ses fils dans l'arbre de réflexion-réfraction-ombre, qui était destiné justement au processeur P. Si les résultats concernant le rayon i sont bien tous revenus au processeur P qui avait créé le rayon i, il transmettra le résultat au processeur hôte qui contrôle l'accès à la mémoire d'image.

La différence entre un processeur à l'intérieur d'un cube et un processeur lié à l'écran est que le premier a beaucoup moins de travail à accomplir : il doit soit calculer des intersections et le cas échéant relancer des rayons secondaires, soit transmettre les messages qui ne lui sont pas destinés, dans la bonne direction. Il n'a donc aucun calcul d'intensité à faire.

Pour que le processeur qui doit calculer une intensité soit sûr que tous les résultats sont bien arrivés, l'arbre ayant une profondeur variable et n'étant pas complet, il faut un compteur qui chaque fois que des rayons secondaires sont lancés, sera incrémenté et donné en héritage aux rayons secondaires.

En conclusion, sans rentrer dans les détails des calculs effectués lors de la simulation de ces deux réseaux, on peut dire que, bien qu'on ne puisse réellement estimer dans l'absolu les temps d'exécution, il est vraisemblable que si le nombre de processeurs est largement inférieur au nombre total de rayons, c'est le temps passé par le processeur le plus occupé qui sera une bonne mesure, sinon, si le nombre de processeurs est largement supérieur au nombre de rayons (ce qui en l'état de la technique n'est pas encore très envisageable), c'est le temps du plus long rayon qui sera à prendre en compte. D'après les simulations, les améliorations en temps sont moins que linéaires en nombre de processeurs s'il y a plus de dix processeurs.

Depuis ces dernières années de nouvelles techniques de parallélisation ont été proposées ([NISH83], [DEGU84], [ENGL88], [FORG87], [KOBA87], [KOBA88], [BOUA88]) ainsi que des VLSI dédiés au tracé de rayons ([NARU87], [TAKA87], [PULL87]). Mais aucune solution satisfaisante aux problèmes posés par la parallélisation n'a été proposée. En effet, il n'existe pas encore d'algorithme qui permette une répartition égale (ou presque égale) des charges de travail pour tous les processeurs. De plus, le découpage de la base de données entraîne une multiplication des échanges de message (ainsi que leur taille). Trop d'échanges de message diminue les gains de temps. Il parviendra à un processeur gérant une zone de l'espace qui contient une source lumineuse, beaucoup plus de messages ; il aura donc beaucoup plus de calculs à faire, on peut ainsi arriver à une situation de blocage. Les recherches en cours semblent se diriger vers un équilibre dynamique des charges des processeurs.

7. Perspectives

Selon les critères actuels, le tracé de rayons exige une très grande puissance de calcul. Ce défaut longtemps rédhibitoire a retardé l'emploi du tracé de rayons, mais la construction de machines spécialisées devrait résoudre le problème des performances.

La quête du réalisme est-elle donc achevée ? Certes non ! Déjà, l'infographie s'attaque aux problèmes mal résolus par le tracé de rayons :

Une surface peut être illuminée indirectement, par l'intermédiaire d'un miroir ou d'une lentille. Le tracé de rayons va émettre un rayon vers la source, mais comment pourrait-il "deviner" le rayon indirect qui passe par le miroir ou la lentille ? Ce problème ne peut être résolu efficacement que pour des miroirs ou des lentilles de forme simple [ARVO86]. La seule solution générale doit lancer énormément de rayons, pour qu'au moins un des rayons lancés à partir d'une surface illuminée indirectement "trouve" la lentille ou le miroir.

De même, le tracé de rayons ne peut simuler correctement les effets de lumières diffuses, notamment dans les scènes d'intérieur, que par un "super-échantillonnage" ("supersampling") terriblement inefficace. Le concept de radiosité a été introduit pour prendre en compte les effets complexes dûs aux réflexions diffuses : [GORA84], [COHE85], [NISH85], [COHE85], [COHE86], [NISH86], [GREE86], [IMME86], [WALL87], [WARD88]...

Chapitre 2
Optimisations

1 Introduction

Depuis maintenant huit ans de nombreux chercheurs se sont penchés sur le problème de l'efficacité du tracé de rayons. Cette méthode de visualisation est très séduisante du fait de sa simplicité à mettre en œuvre presque tous les phénomènes optiques classiques : ombres, réflexions, réfractions. Mais une telle simplicité a un lourd tribut à payer : le temps de calcul nécessaire. Effectivement, pour chaque pixel de l'image, un traitement est nécessaire ; ce qui signifie que ce temps de traitement, même s'il est très court, devra être multiplié par 2^{18} pour une image de taille 512x512, en ne comptant que les rayons issus de l'œil.

Les optimisations proposées sont plus ou moins bonnes selon le type de rayons lancés. En effet, leurs caractéristiques(primaires, d'ombres ou de réflexions/réfractions) entraînent que certaines optimisations excellentes pour les rayons primaires, par exemple, sont inadaptées aux rayons de réflexions/réfractions. Car ces optimisations utilisent ou, au contraire, n'utilisent pas assez le fait que :

- tous les rayons primaires partent d'un même point : l'œil,
- tous les rayons d'ombres, émis vers le soleil, ont tous la même direction, tous les rayons d'ombres, émis par une source lumineuse à distance finie, arrivent tous à cette source lumineuse,
- les rayons de réflexions/réfractions sont distribués dans l'espace, sans direction privilégiée.

Nous nous proposons par conséquent d'utiliser un type d'optimisation lié au type de rayon afin d'avoir les meilleures performances. Nous allons tout d'abord, en tenant compte des caractéristiques des rayons, étudier les optimisations déjà proposées, en précisant celles que nous éliminons et celles que nous prendrons comme base :

- projection sur l'écran ([ROTH82], [WEGH84], [BRON84], [GERV86], [EXCO87], [VERR87],[ARVO87]). Cette méthode est tout à fait efficace pour les rayons primaires. Elle a été étendue en utilisant des écrans fictifs pour le cas des sources lumineuses ponctuelles [HAIN86]. Nous nous baserons sur elle en l'étendant à l'utilisation d'arbre C.S.G.
- projection sur le plan horizontal ([COQU84]). Cette méthode est efficace pour la visualisation de terrain, mais elle pose des problèmes lorsque plusieurs objets se projettent sur la même cellule. En effet, elle est moins bonne que la méthode de projection sur l'écran pour les rayons primaires lorsque la scène est quelconque.

- volumes englobants ([WITH80], [RUBI80], [ROTH82], [WEGH84], [KAY86], [GLAS88]). L'utilisation de volumes englobants est valable que ce soit pour les rayons primaires ou secondaires. Cette utilisation n'est absolument pas incompatible avec notre méthode.

- octrees, voxels, bintrees ou modèles hiérarchiques ([RUBI80], [FUJI86] [GLAS84], [KUNI85], [AMAN87], [PENG87], [SCHE87]) Ces structures peuvent être utilisées pour les rayons primaires et secondaires avec l'inconvénient de leur généralité qui ne prend pas en compte les particularités des rayons primaires ou d'ombres. Elles sont, en revanche, adaptées aux rayons de réflexions/réfractions. Mais la place nécessaire pour avoir une bonne définition, sans laquelle cette méthode ne présente plus guère d'intêret, est rédhibitoire ; ce n'est donc pas la solution retenue.

- structures de cohérence spatiale ([ARNA87], [BOUA87]). La structure qu'ils proposent est une sorte d'octree irrégulier dont les cellules ont la forme de pyramide tronquée. Ils se basent sur la projection sur l'écran des boîtes englobantes de l'arbre CSG représentant la scène pour construire cette structure. En effet la pyramide de la structure totale a pour sommet le plan de l'écran et pour base le plan parallèle à l'écran de la boite englobante la plus éloignée de l'œil. Chaque cellule de cette structure est connectée par des pointeurs à ses voisines, ce qui est relativement compliqué à gérer. Les rayons primaires n'ont en réalité besoin que d'une connexion en profondeur. Tandis que les rayons secondaires peuvent prendre une direction arbitraire dans cette structure et l'algorithme de passage d'une cellule à une autre n'est pas aussi performant que ceux proposés pour des octrees [KUNI85], [AMAN87]. Les auteurs ne présentent pas la technique qu'ils utilisent lorsque les objets se trouvent derrière l'œil. En effet, si ces objets ne sont pas visibles pour les rayons primaires, ils peuvent projeter de l'ombre ou se réfléchir dans les objets visibles. De plus, la complexité de l'arbre BSP qu'ils utilisent ne dépend que de la complexité de la scène, et ne dépend pas de la taille de l'image : est-il utile de subdiviser une région de l'écran qui ne couvre qu'un pixel ? Mais cette méthode, après initialisation des structures, a une complexité presque indépendante du nombre d'objets. La deuxième méthode qu'ils proposent ne travaille plus sur une projection sur l'écran mais directement par une subdivision par une maille régulière dans l'espace de la scène (cf Chapitre 1 §4.1.5).

- méthodes spécifiques à la représentation des objets : surfaces Bsplines [YANG87], surfaces de Bézier [JOY 86], tesselations [SNYD87].

- les méthodes de sous-échantillonnage, [ROTH82], [BRON84], peuvent entraîner une dégradation de la qualité de l'image.

- les méthodes de cohérence : la méthode de cohérence de Speer et al.,[SPEE85], apporte, selon les auteurs, peu de gain et d'autres techniques d'optimisations

doivent être utilisées. Peachey, [PEAC86], utilise la cohérence pour éviter de retester des boîtes englobantes déjà étudiées par les rayons précédents ; il obtient un gain d'environ 10 %. Ohta et al., [OHTA87], construisent à partir des volumes englobants autour de l'œil, des sources lumineuses et des objets réfléchissants ou réfractants, une table contenant, pour chaque région ou secteur d'un de ces volumes englobants, l'ensemble des objets qui sont vus. Cela donne, apparemment, une visualisation rapide en tracé de rayons, mais les quantités de mémoire nécessaires sont énormes quand on a beaucoup d'objets produisant des rayons secondaires. Cette méthode revient, en quelque sorte, à associer à chaque "objet" à problème, œil et sources lumineuses comprises, un écran virtuel qui les entoure. Ce qui nous ramène, plus ou moins, à la première méthode exposée.

- les méthodes qui, à partir d'une description de la scène par un arbre de construction, obtiennent des objets polyédriques, [ATHE83], [BRON86], [VERR87]. Ces méthodes produisent des images d'une qualité dégradée (les sphères sont polyédriques), et ne sont valables que pour les rayons primaires.

Le classement que nous venons de faire des différentes méthodes d'optimisation, fait intervenir certaines de ces méthodes dans plusieurs classes. C'est une conséquence de la productivité en ce domaine. En effet, les premières techniques proposées pouvaient être facilement classées. Mais, devant la finesse des études qui ont été faites, il devient de plus en plus difficile de cataloguer ces méthodes qui sont des rejetons hybrides des techniques de bases. Nous n'échapperons pas à ce cas de figure. Les optimisations que nous proposons sont des extensions originales, mais se reposent sur des travaux déjà exposés, pour les rayons primaires ou secondaires. Nous exposerons la technique de [BRON84] en détail, car elle se rapproche le plus de notre méthode. Mais nous améliorons ses performances. En revanche, nous n'avons trouvé dans aucune des publications sur le tracé de rayons (pourtant nous référençons environ 70 articles sur ce domaine), des optimisations basées sur ce que nous appellerons l'instanciation locale.

Donc, sans remettre en cause l'utilisation de volumes englobants qui sont utiles pour toutes les sortes de rayons, nous proposons une méthode spécifique à chacun des types de rayons. Nous allons, par conséquent, proposer quatre optimisations :

1 pour les rayons primaires,

2 pour les rayons d'ombre,

3 pour les rayons de réflexions/réfractions,

4 pour améliorer les temps de calculs d'intersection entre un rayon et un arbre de construction.

La dernière partie de ce chapitre abordera les problèmes d'imprécision et d'ambiguité que pose l'utilisation d'un modèle CSG. Puis nous présenterons quelques

extensions du modèle CSG résolvant ces problèmes et une extension permettant quelques optimisations. Tous les résultats : gains de temps et statistiques, seront regroupées en annexe.

2 Pré-supposé

Nous travaillons sur une description de la scène par un arbre CSG[1] que nous supposerons binaire. Un nœud de l'arbre est une opération ensembliste : union, intersection ou différence qui opère sur le sous-arbre gauche et le sous-arbre droit de l'arbre. Les feuilles sont composées d'une primitive élémentaire : sphère, cube, cylindre, prisme, cône ... et d'une matrice de passage du repère de la primitive élémentaire au repère de la scène. A chaque nœud de l'arbre est attachée une boîte et une sphère englobant les sous-arbres gauche et droit. Cette boîte englobante dont les faces sont parallèles aux plans définis par les axes du repère de la scène (par abus de langage on dira que cette boîte est parallèle aux axes de la scène bien que le terme exact soit isothétique) est déduite récursivement de la boîte englobante du sous-arbre gauche et de la boîte englobante du sous-arbre droit, modulo l'opération ensembliste associée au nœud considéré. Le choix d'une boîte englobante parallèle aux axes de la scène a été dicté par la simplicité de conception et d'utilisation, de plus, ces boîtes englobantes sont utilisées dans chacune des structures d'optimisation des rayons primaires et secondaires.

A chaque nœud et à chaque feuille de l'arbre est attaché un drapeau qui indique si, sur ce nœud (ou cette feuille), seront appliquées des opérations d'intersection ou de différence. En effet, lorsqu'une sous-arborescence complète ne contient que des unions, (nous appellerons sous-arborescence complète tout sous-arbre dont la racine est celle de l'arbre dont il est tiré et dont tous les chemins issus de la racine mènent à une feuille contenant une primitive élémentaire), pour toutes les primitives contenues dans cette sous-arborescence complète on n'a besoin de connaître que les points d'entrée.

---

[1] cf chapitre 1 paragraphe 2.1

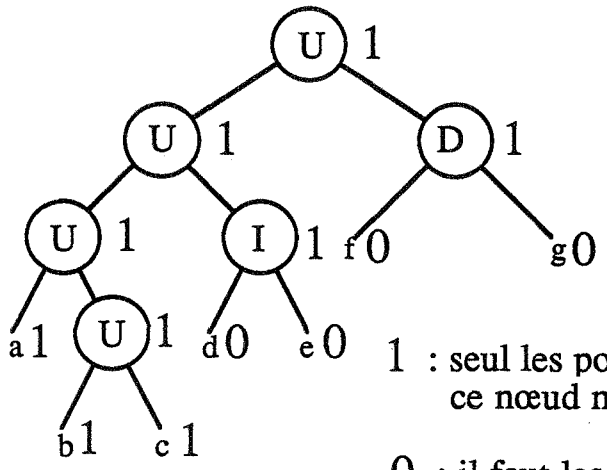


Fig. 1 Exemple d'arbre de construction

De plus, le point le plus proche suffit pour trouver l'intersection d'un rayon primaire ou d'ombre avec la scène. Donc dans ce cas il est inutile de trier les points d'intersections, le minimum (au sens de la distance par rapport à l'œil) est suffisant.

3 Rayons primaires

Nous allons tout d'abord présenter la méthode de Bronsvoort possédant quelques similitudes avec notre méthode. Puis nous introduirons le principe du balayage du plan. Cette méthode nous permettra de déterminer les intersections entre les rectangles englobants calculés à partir de la projection des boîtes englobantes sur le plan de l'écran. En effet, lors de la construction récursive des boîtes englobantes de chaque nœud de l'arbre C.S.G., les points définissant cette boîte sont projetés sur le plan de l'écran. Ce qui permet de définir des rectangles englobants (à coordonnées entières, et dont les cotés sont parallèles aux bords de l'écran) qui regroupent les pixels de l'écran en zone d'influence des objets de la scène. Nous voulons déterminer pour chaque zone rectangulaire de l'écran qu'elle est la scène minimale qui s'y projette. Pour cela nous allons effectuer directement sur les rectangles les opérations booléennes associées aux nœuds de l'arbre de construction (cf. figure suivante).

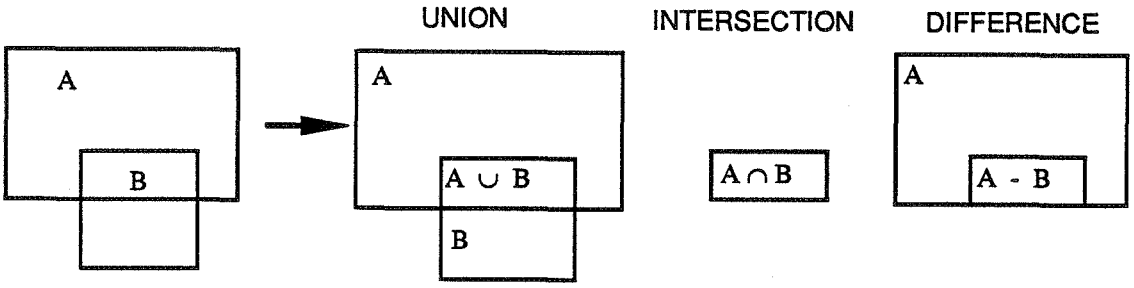


Fig. 2 Résultats d'opérations booléennes

Bronsvoort et al., [BRON84], associent à chaque nœud de l'arbre de construction, des intervalles englobants sur l'écran, au lieu de boîtes englobantes. En effet ils ont remarqué que si deux objets sont combinés à l'aide d'une opération union, la boîte englobante de l'union de ces deux objets ne donne pas une bonne indication du volume occupé (cf Fig. 3).

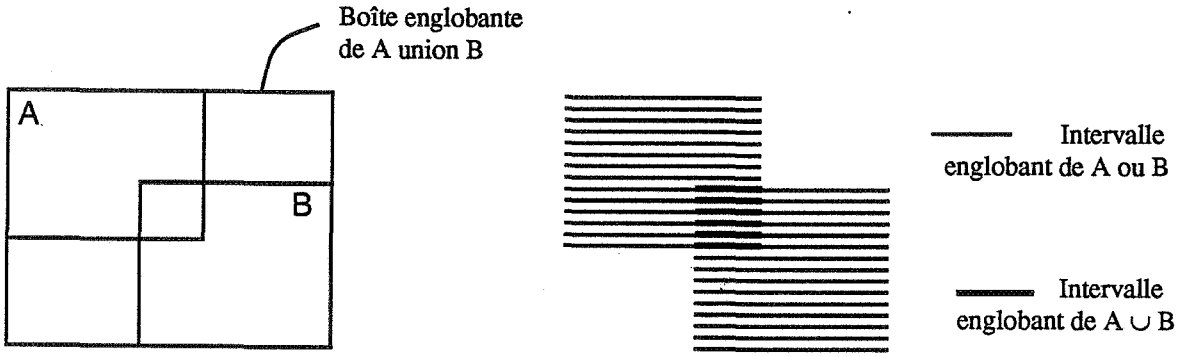


Fig.3 Rectangles englobants versus intervalles englobants

Au lieu d'utiliser un test d'intersection entre un rayon et une boîte, ils emploient un test d'appartenance d'un point à une intervalle sur l'écran. Pour calculer une image, ils utilisent un balayage de lignes et déterminent pour chaque ligne les intervalles englobants.Cet intervalle pourra changer d'une ligne à l'autre à chaque nœud de l'arbre de construction est associé un intervalle qui correspond à l'intervalle actif sur l'écran pour une ligne de balayage donnée. Pour gagner du temps, ils "tronquent" l'arbre de construction en utilisant en plus des pointeurs sur les sous-arbres gauche et droit, des pointeurs qui désignent directement les sous-arbres réellement actifs dans cette ligne de balayage (cf Fig. 4).

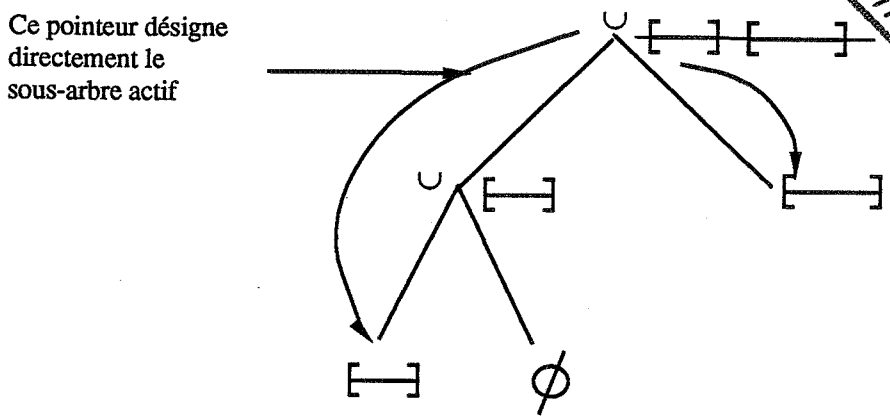


Fig. 4 Simplification de l'arbre de construction

Mais dès que les intervalles changent d'une ligne à la suivante, ils doivent recalculer les intervalles englobants de la scène et reconstruire l'arbre réduit, ce qui est pénalisant. Nous voulons lors d'un pré-calcul déterminer non seulement des rectangles englobants minimaux mais aussi une sous-scène active minimale pour chacun des rectangles. Le but que nous nous fixons est d'associer, de façon optimale, à chaque zone rectangulaire de l'écran, le sous-arbre CSG minimal qui se projette sur cette zone. Il convient donc de trouver les intersections entre les rectangles englobants. Pour faire cette recherche, on utilise le principe du balayage du plan.

3.1 Algorithme du balayage du plan

3.1.1 Historique

Shamos et Hœy ont introduit la technique du balayage du plan dans [SHAM76], afin d'obtenir un algorithme optimal pour tester l'existence d'un point d'intersection parmi un ensemble de segments. Bentley et Ottmann proposent, dans [BENT79], une modification de l'algorithme précédent pour calculer tous les points d'intersection. Enfin, Niervegelt et Preparata exposent, dans [NIER82], deux nouvelles applications de la méthode de balayage du plan pour la décomposition d'un polygone croisé et pour l'intersection de deux cartes planaires. Tallot expose, dans [TALL85] de façon détaillée l'algorithme du balayage du plan et propose un algorithme optimal quand le nombre de directions des segments est faible. Gangnet et Michelucci [GANG84] construisent à partir de segments une carte planaire locale, en utilisant l'algorithme de Bentley-Ottmann, puis déduisent, de la carte planaire locale, la carte planaire globale. Ces cartes planaires ont été utilisées, entre autres, pour des recalages, désignations et affectations sur des croquis d'architecture, saisies à main levée, pour des logiciels d'animation [MOIS84], pour une optimisation du tracé de rayons [COQU84], dans un algorithme de visualisation de scènes 3D.

3.1.2 Principe de Bentley-Ottmann

On a, pour données, n segments. On trie les extrémités de ces segments dans l'ordre lexicographique (que nous appelerons xy-ordre), c'est à dire, si 0xy est le repère dans lequel sont exprimées les coordonnées des extrémités, elles seront classées par x croissant, et pour des x égaux, par y croissant. On utilise une droite de balayage parallèle à 0y. Cette droite D_i va passer par toutes les extrémités (x_i,y_i) des segments en suivant l'xy-ordre. Pour $D_i = \{(x,y),\ x = x_i\}$, on possède un ordre d'adjacence entre les segments actifs. On appelle segment actif, un segment dont la première extrémité (par rapport à l'xy-ordre), est avant la droite D_i et la deuxième est après D_i. Cet ordre permet de ne tester l'intersection qu'entre les segments dont l'ordre d'adjacence a changé entre D_{i-1} et D_i. Si on trouve des intersections, elles sont ajoutées dans la structure en tenant compte de l'xy-ordre. Cet algorithme a une complexité théorique de O ((n+k)log(n)), où k est le nombre d'intersection. Dans le pire des cas, k est de l'ordre de n^2.

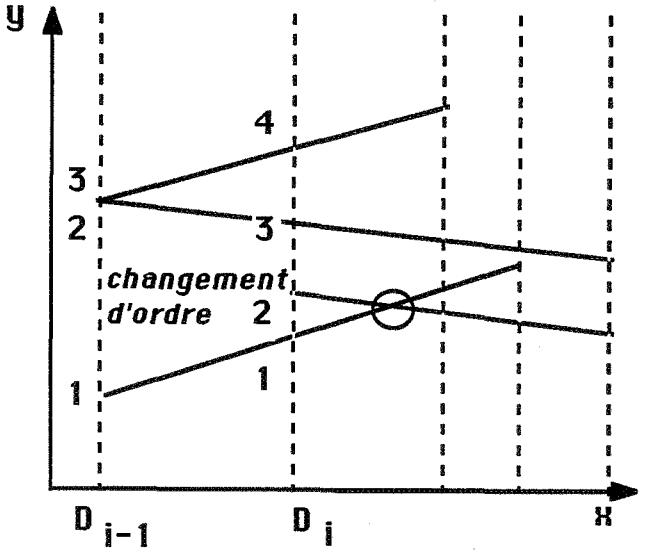


Fig 5. Balayage du plan.

3.2 Intersection entre rectangles avec un calcul de sous-scène associée

Pour appliquer l'algorithme de Bentley-Ottmann directement, les arêtes des rectangles englobants doivent être triées par ordre lexicographique. Ensuite, on peut trouver les intersections entre les arêtes. Mais il faudrait alors retrouver l'information topologique pour créer de nouveaux sous-rectangles, de plus certaines intersections sont inutiles. Par exemple dans le cas où l'arbre comprend une différence : (A-B) $\cup$ C, on a pas besoin de connaître l'intersection entre B-A et C. Retrouver toutes ces informations à partir de la liste des segments et des points d'intersection n'est pas très simple. Nous allons par conséquent utiliser une autre méthode parcourant de façon récursive l'arbre CSG.

```
rectangles( scène, liste_rectangles)
{
si scène est une feuille
    liste_rectangles = rectangle_de_la_primitive( scène);
sinon /* scène est un nœud */ {
    rectangles( scène->filsg, listg);
    rectangles( scène->filsd, listd);
    gérer_list_en_x( listg, listd, liste_rectangles,scène->op);
    }
}
```

La procédure *gérer_list_en_x* retourne une liste de rectangles ne s'intersectant pas et lexicographiquement triées à partir de deux listes de rectangles triées dont les rectangles (pour chacune des listes) ne s'intersectent pas, en calculant les intersections de rectangles modulo l'opération booléennes op associée au nœud. Nous allons maintenant décrire de façon plus précise la procédure *gérer_list_en_x*. A une étape donnée nous ayons deux listes de rectangles lexicographiquement triées (cf. Fig. 6 le résultat de l'appel de la procédure sur ces deux listes se trouve Fig. 8) ; à chacun des rectangles est associée une sous-scène, et un opérateur ensembliste op. On suppose, aussi, que dans chaque liste, les rectangles ne s'intersectent pas. Il faut donc trouver l'intersection entre ces deux listes de rectangles. Le résultat que l'on veut obtenir, c'est la liste des rectangles issus du calcul d'intersection et de l'opération ensembliste op.

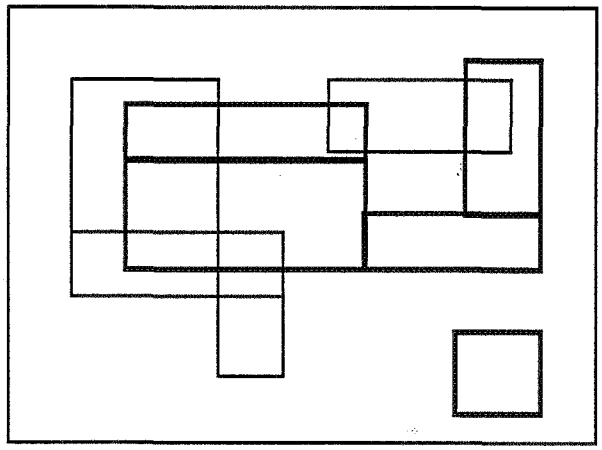


Fig. 6 Deux listes de rectangles liées par un opérateur différence (cf Fig. 8)

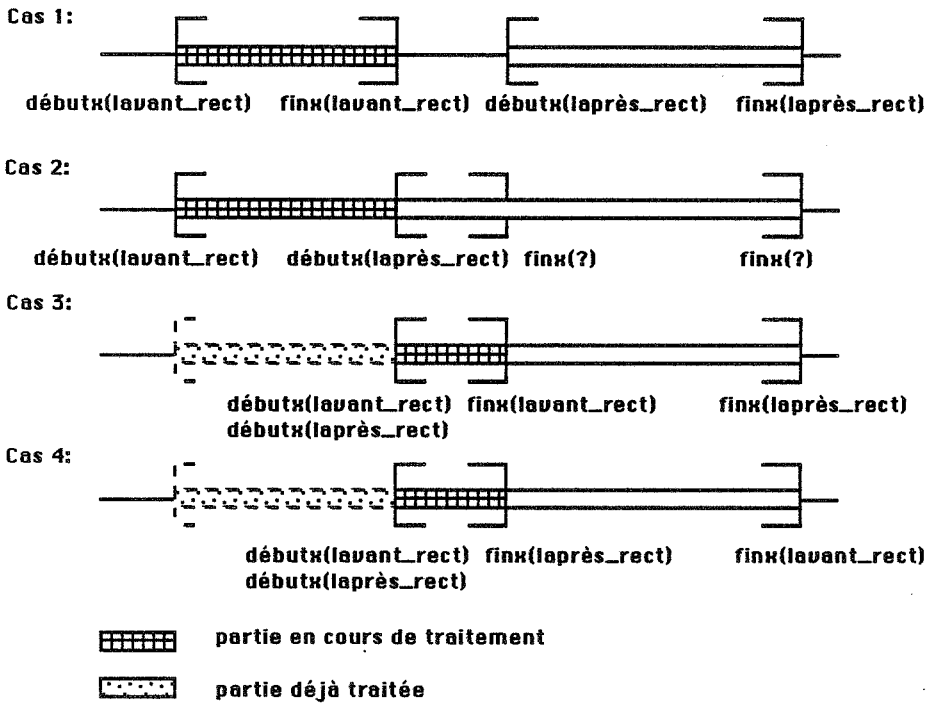


Fig. 7 Projection des rectangles sur l'axe 0x.

Nous proposons l'algorithme suivant :

```
gérer_list_en_x(listg, listd, liste_de_sortie,op)
TANTQUE la liste gauche, listg, ou la liste droite, listd, n'est pas vide
   {
   lrectg est la première liste des rectangles de listg qui ont la même origine en x;
   lrectd est la première liste des rectangles de listd qui ont la même origine en x;
   avant_rect est égal à lrectg, si lrectg est devant, en x, lrectd, à lrectd sinon;
   après_rect est égal à lrectg, si lrectg est après, en x, lrectd, à lrectd sinon;
   débutx est une fonction qui donne la plus petite des coordonnées en x d'une liste
                de rectangles;
   finx est une fonction qui donne la plus grande des coordonnés en x d'une liste
                de rectangles;
   SI (finx(avant_rect) ≤débutx(après_rect) ) (cf fig. 5cas 1)
    /*************************************************************/
      les deux listes de rectangles lrectg, lrectd, ne s'intersectent pas;
      selon l'opérateur on ajoute ou non avant_rect dans la liste_de_sortie;
      on avance dans la liste associée à avant_rect en supprimant tous les rectangles
                contenus dans avant_rect;
   SINON
      SI (débutx(avant_rect) ≠ débutx(après_rect) ) (cf fig. 5 cas 2)
      /*********************************************************************/
         on recopie avant_rect dans une_liste en donnant comme bornes en x
                débutx(avant_rect) et débutx(après_rect);
      selon l'opérateur on ajoute ou non une_liste dans la liste_de_sortie;
      on tronque avant_rect afin que débutx(avant_rect) soit égal à débutx(après_rect);
      FINSI
         SI ( finx(avant_rect) ≤ finx(après_rect) ) (cf fig. 5 cas 3)
         /*************************************************************/
            on recopie lrectg dans une_listg, et lrectd dans une_listd en donnant
                comme bornes en x débutx(après_rect) et finx(avant_rect);
            on réapplique le même genre d'algorithme sur une_listg et une_listd
                mais en travaillant en y et non plus en x :
            gerer_list_en_y(une_listg,une_listd,liste_de_sortie,op); (1)
            on tronque après_rect afin que débutx(après_rect) soit égal à
                finx(avant_rect);
            on avance dans la liste associée à avant_rect en supprimant tous les
                rectangles contenus dans avant_rect;
         SINON (cf fig. 5 cas 4)
          /**********************/
            on recopie lrectg dans une_listg, et lrectd dans une_listd en donnant
                comme bornes en x débutx(après_rect) et finx(après_rect);
            on réapplique le même genre d'algorithme sur une_listg et une_listd mais
                en travaillant en y et non plus en x :
            gerer_list_en_y(une_listg,une_listd,liste_de_sortie,op); (1)
            on tronque avant_rect afin que débutx(avant_rect) soit égal à
                finx(après_rect);
            on avance dans la liste associée à après_rect en supprimant tous
                les rectangles contenus dans après_rect;
         FINSI
   FINSI
FINTANTQUE
```

```
gerer_list_en_y( listg, listd, liste_de_sortie,op)

TANTQUE la liste gauche, listg, ou la liste droite, listd, n'est pas vide
   {
   rectg est le premier rectangle de listg en y;
   rectd est la premier rectangle de listd en y;
   avant_rect est égal à rectg, si rectg est devant rectd, en y, sinon à rectd;
   après_rect est égal à rectg, si rectg est après rectd, en y, sinon à rectd;
   débuty est une fonction qui donne la plus petite des coordonnés en y d'un rectangle;
    finy estune fonction qui donne la plus grande des coordonnés en y d'unrectangle
   SI ( finy(avant_rect) ≤ débuty(après_rect) ) (idem fig. 5cas 1)
   /*****************************************************************/
      les deux rectangles rectg, rectd, ne s'intersectent pas;
      selon l'opérateur op on ajoute ou non avant_rect dans la liste_de_sortie;
      on avance dans la liste associée à avant_rect en supprimant le rectangle
                        avant_rect;
   SINON
      SI ( débuty(avant_rect) ≠ débuty(après_rect) ) (idem fig. 5 cas 2)
      /*************************************************************************/
         on recopie avant_rect dans un_rect en donnant comme bornes en y
                        débuty(avant_rect) et débuty(après_rect);
         selon l'opérateur op on ajoute ou non un_rect dans la liste_de_sortie;
         on tronque avant_rect afin que débuty(avant_rect) soit égal à
                        débuty(après_rect);
      FINSI
      SI ( finy(avant_rect) ≤ finy(après_rect) ) (idem fig. 5 cas 3)
      /*****************************************************************/
         on recopie avant_rect dans un_rect en donnant comme bornes en y
                        débuty(après_rect) et finy(avant_rect);
         on met dans un_rect la scène résultant de l'opération ensembliste op entre la
                        scène associée à rectg et celle associée à rectd ;
         on ajoute un_rect à la liste_de_sortie.
         on tronque après_rect afin que débuty(après_rect) soit égal à finy(avant_rect);
         on avance dans la liste associée à avant_rect en supprimant le rectangle
                        avant_rect;
      SINON (idem fig. 5 cas 4)
      /************************/
         on recopie avant_rect dans un_rect en donnant comme bornesen y
                        débuty(après_rect) et finy(après_rect);
         on met dans un_rect la scène résultant de l'opération ensembliste op entre la
                        scène associée à rectg et celle associée à rectd;
         on ajoute un_rect à la liste_de_sortie.on tronque avant_rect afin que
                        débuty(avant_rect) soit égal à finy(après_rect);
         on avance dans la liste associée à après_rect en supprimant le rectangle
                        après_rect;
      FINSI
   FINSI
FINTANQUE
```

L'algorithme d'intersection sur l'axe 0y est identique au précédent sauf dans les cas 3 et 4 où, l'instruction (1) est remplacée par la construction de la scène associée au rectangle d'intersection, en faisant l'opération ensembliste op sur les deux sous-scènes associées aux deux rectangles initiaux. Puis on ajoute ce nouveau rectangle à la liste de sortie.

3.3 Etude de la complexité
3.3.1 Calcul de la complexité

On obtient comme résultat une liste de rectangles lexicographiquement triée. On appelle la procédure *gest_diff_list* récursivement sur chacun des nœuds de l'arbre de construction. Le calcul de complexité d'un tel algorithme dépend du nombre de rectangles résultant du calcul d'intersection. Ce nombre K_n peut être supposé compris entre n et n^2, où n est le nombre de primitives (donc de rectangles initiaux). Nous discuterons un peu plus loin, de façon plus fine la valeur de K_n. Nous supposerons tout d'abord que l'arbre de construction est équilibré et que $n = 2^p$. Soit P(n) la complexité de l'algorithme, nous avons :

$$P(n) = P_G(n) + P_D(n) + K_n$$

$$P(n) = 2\ P(\frac{n}{2}) + K_n$$

$$P(n) = 2^p\ P(1) + \sum_{i=0}^{i=p} 2^{p-i}\ K_2{}^i$$

Si $K_i = c\ i$, où c est une constante, on obtient :

$$P(n) = 2^p\ P(1) + \sum_{i=0}^{i=p} c\ 2^{p-i+i}$$

$$P(n) = 2^p\ P(1) + \sum_{i=0}^{i=p} c\ 2^p$$

$$P(n) = 2^p\ P(1) + c\ \ 2^p\ p$$

$$P(n) = n\ P(1) + c\ n\ \ \log(n)$$

donc $P(n) \approx O\ (n\log(n))$.

Si $K_i = c\ i^2$, on obtient :

$$P(n) = 2^p\ P(1) + \sum_{i=0}^{i=p} c\ 2^{p-i+2i}$$

$$P(n) = 2^p\ P(1) + \sum_{i=0}^{i=p} c\ 2^{p+i}$$

$$P(n) = 2^p\ P(1) + c\ p \sum_{i=0}^{i=p} 2^i$$

$$P(n) = n\ P(1) + c\ n\ (2n-1)$$

donc $P(n) \approx O\ (n^2)$.

Nous avons, par conséquent des ordres de grandeur compris entre O (nlogn) et $O\ (n^2)$ pour un arbre équilibré. Si nous prenons, maintenant, l'exemple d'un arbre peigné :

$$P(n) = P(n-1) + K_n$$

$$P(n) = P(1) + \sum_{i=0}^{i=p} K_i$$

Si $K_i = c\ i$, on obtient :

$$P(n) = P(1) + \sum_{i=0}^{i=p} c\ i$$

$$P(n) = P(1) + \frac{c\ n\ (n-1)}{2},$$

donc $P(n) \approx O\ (n^2)$.

Si $K_i = c\ i^2$, on obtient : $P(n) \approx O\ (n^3)$.

3.3.2 Démonstration de son optimalité

Bentley et Wood ont proposé un algorithme optimal de recherche de paires de rectangles intersectants [BENT80]. Le principe de leur algorithme est le suivant :

[1] Ils cherchent toutes les paires de rectangles dont les cotés s'intersectent en utilisant l'algorithme de Bentley-Ottmann.

[2] Ils cherchent toutes les paires de rectangles dont l'un est entierement inclus dans l'autre. Pour faire cela, ils commencent par générer un ensemble de points inclus dans les rectangles (le centre de chaque rectangle, par exemple). Ensuite, pour chacun des points, ils trouvent l'ensemble des rectangles qui le contiennent. Puis, ils comparent chaque rectangle de la liste avec le rectangle associé à ce point. Si ces rectangles ont des arêtes intersectantes, il n'y a rien à faire. Sinon, l'un des rectangles est inclus dans l'autre.

Ils démontrent que cet algorithme est optimal en O (nlogn+k), où k est le nombre de paires de rectangles intersectants. Maintenant, si nous voulons pour chaque paire de rectangles intersectants construire la scène correspondante, nous devons parcourir la scène initiale pour accéder aux feuilles et ainsi faire les opérations ensemblistes. Il est nécessaire d'aller chercher cette information aux feuilles de l'arbre. Soit f(n) la complexité pour accéder à une feuille de l'arbre, ($f(n) = O$ (log(n)) si l'arbre est équilibré, sinon dans le pire des cas, $f(n) = O$ (n)). La complexité d'un tel algorithme est donc de O (nlogn+kn), c'est un algorithme optimal. Or k est du même ordre que K_n, les deux algorithmes ont par conséquent la même complexité. L'algorithme proposé est optimal.

3.3.3 Réflexion sur cette complexité

La complexité, dans le pire des cas, devient assez mauvaise. Mais on peut tout d'arbord remarquer que le cas où $K_n = k\ n^2$ est très rare, sauf si on considère une mine à ciel ouvert où tous les rectangles s'emboîtent comme des "poupées russes". De plus, on supprime les rectangles qui n'interviennent pas dans l'arbre de construction (cf. Fig. 8).

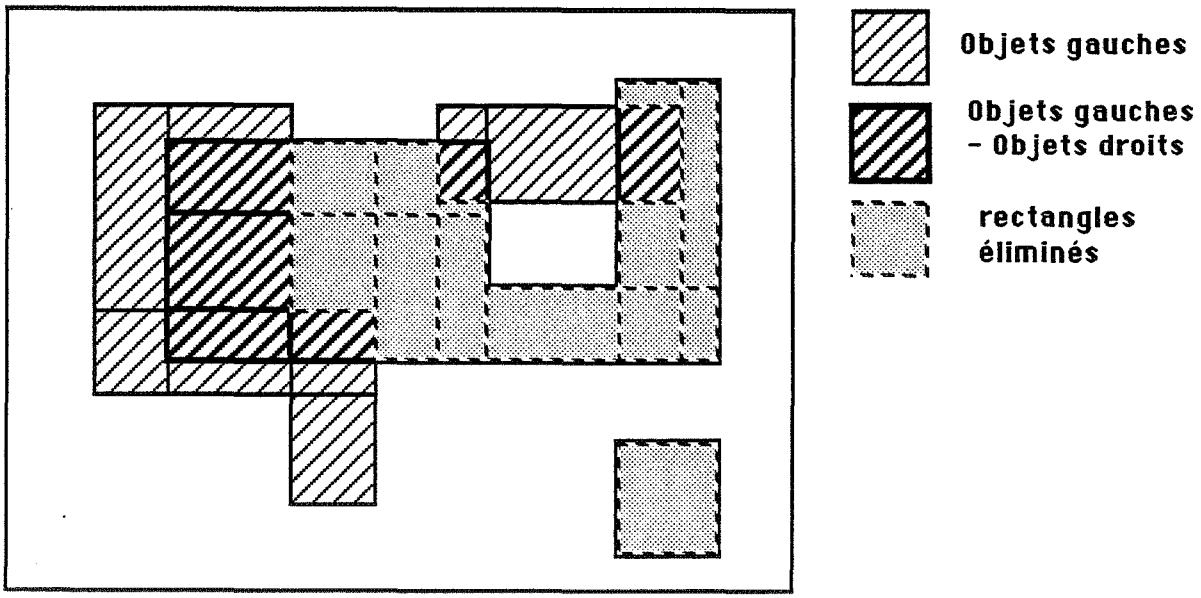


Fig. 8 Résultat d'un calcul d'intersection avec un opérateur différence (cf Fig. 6).

Par conséquent K_n peut même être inférieur à n. De plus pour une scène composée uniquement d'unions, il n'est pas vraiment nécessaire de chercher les rectangles d'intersection ; on peut faire simplement une sorte de tampon en z sur les rectangles. Les rectangles sont affichés les uns après les autres dans la structure de stockage, ceux qui s'affichent au même endroit, sont classés du plus proche au plus éloigné par rapport à leur distance à l'œil. Donc la méthode de balayage ne sera utilisée que pour les sous-arborescences qui ont une racine différence ou intersection mais qui n'interviennent que dans des unions (nous rappellons que ces sous-arborescences sont déterminées lors de la lecture de la scène c'est à dire en $O(n)$). Si on a s sous-arborescences, si l'on suppose, à titre d'indication, que chacune des sous-arborescences contient le même nombre de primitives, c'est à dire n/s primitives, on obtient une complexité dans le pire des cas de $O(\frac{n^3}{s^2})$.

Toutes les sous-arborescences complètes issues des sous-scènes contenues dans la structure de stockage, sont placées afin que le sous-arbre gauche soit devant le sous-arbre

droit par rapport à l'œil. Dans ce cas, si un point d'intersection, trouvé dans le sous-arbre gauche, est plus proche le l'œil que le point le plus proche de l'œil dans l'arbre droit, il est inutile de chercher l'intersection entre le rayon et le sous-arbre droit.

S'il y a beaucoup de primitives, si la projection de chaque primitive couvre une grande partie de l'écran et si la taille de l'image est très importante, le temps de précalcul peut être trop grand par rapport au temps gagné par l'optimisation. Pour résoudre ce problème, on peut définir une taille de zone élementaire en dessous de laquelle on ne découpera plus en rectangle.

4 Rayons d'ombres

Nous pouvons utiliser la même technique que pour les rayons primaires. Mais l'accès à une zone doit être direct et non plus séquentiel. De plus comme le montrent Haines et Greenberg[HAIN86], il n'est pas nécessaire d'avoir une très grande définition. Ces données peuvent donc être stockées dans un tableau ou un quadtree (quoiqu'une structure aussi compliquée qu'un quadtree soit l'équivalent d'un marteau pilon pour casser une noix). Pour le soleil (source que l'on considère ponctuelle et à distance infinie et donc donnée par un vecteur directeur), on peut définir un écran virtuel qui lui est associé, les boîtes englobantes des objets de la scène se projettent parallèlement à la direction du soleil. A chaque source ponctuelle à distance finie, on peut associer six écrans parallèles aux trois plans définis par les axes du repère de la scène. Si cette source est en dehors de la scène, un seul écran suffit. On peut associer à ces sources une portée maximale. Dans ce cas, les objets dont la boîte englobante est au-delà de la portée maximale ne seront pas projetés sur les écrans fictifs,ce qui économise beaucoup de place dans le cas d'une scène complexe.

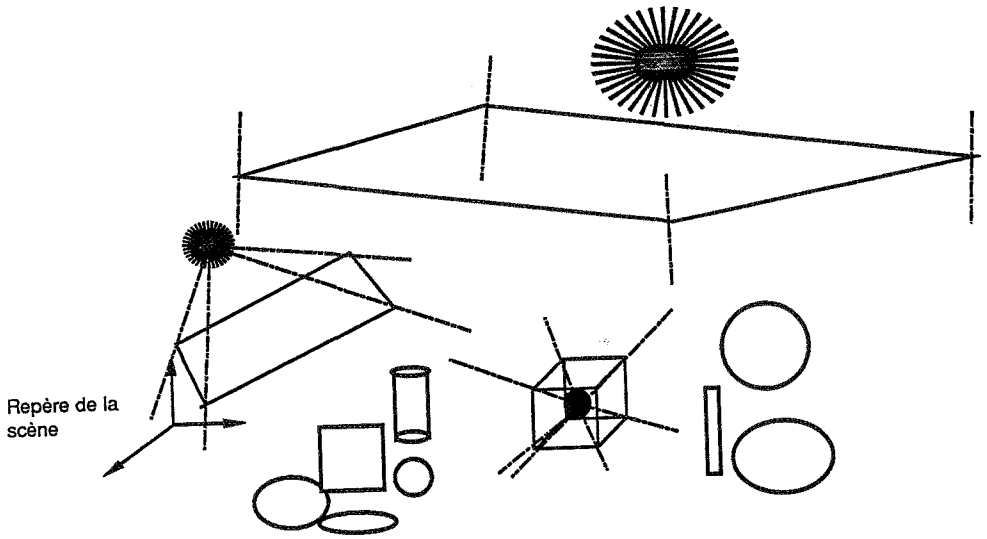


Fig. 9 Ecrans associés aux sources lumineuses

5 Rayons de réflexion-réfraction

Ces rayons ont la particularité d'avoir leur origine incluse dans la boite englobant l'ensemble la scène. Par contre, leur direction est quelconque. L'utilisation d'un plan de projection horizontal n'est intéressant que pour les rayons verticaux. Les octrees, les voxels et les bintrees au caractère aléatoire de ces rayons, mais ils demandent une grande quantité de mémoire. Par conséquent, la plupart du temps, leur définition n'est pas très élevée. Ce qui diminue leur pouvoir de discrimination des objets et rend leur efficacité plus douteuse. En réalité, deux plans de projection suffisent pour reconstruire l'information 3D.

Pour déterminer ces deux plans, une première solution consisterait à étudier un nuage de points, où chaque point est le centre d'une primitive. De ce nuage de point, par une étude statistique on peut déterminer les plans principaux. L'inconvénient de cette méthode, est que l'on perd l'information de volume d'une primitive.

La solution choisie consiste à prendre non pas deux mais trois plans orthogonaux. Ces trois plans sont ceux déterminés par trois des plans de la boîte englobante de la scène. La projection des boites englobantes des objets se fait très simplement : il suffit de prendre deux des trois coordonnées de chacun des sommets de la boîte. Sur chacun de ces plans, on définit un quadtree ; chaque cellule du quadtree contiendra une sous-scène CSG.

Pour chaque rayon, on détermine le plan (et donc le quadtree) qui réduira au mieux le nombre de primitives à tester. En supposant que les primitives sont équidistribuées dans l'espace, c'est le quadtree qui a le moins de cellules à tester. Pour le déterminer, il suffit de connaître la projection du rayon sur chacun des plans. On trouve trois segments, définis par le point de départ du rayon et par le point à partir duquel le rayon sort de la boite englobante de la scène. Le segment le plus court est celui qui traverse le moins de cellules. Supposons le rayon déterminé par (R,u) où R est le point de départ du rayon et u sa direction. Le plan parallèle à Ox sera choisi si $|u_x| > |u_y|$ et $|u_x| > |u_z|$. Le plan parallèle à Oy sera choisi si $|u_y| > |u_x|$ et $|u_y| > |u_z|$. Sinon, le plan parallèle à Oz sera choisi. Donc quatre tests, au plus, suffisent pour déterminer le quadtree à utiliser.

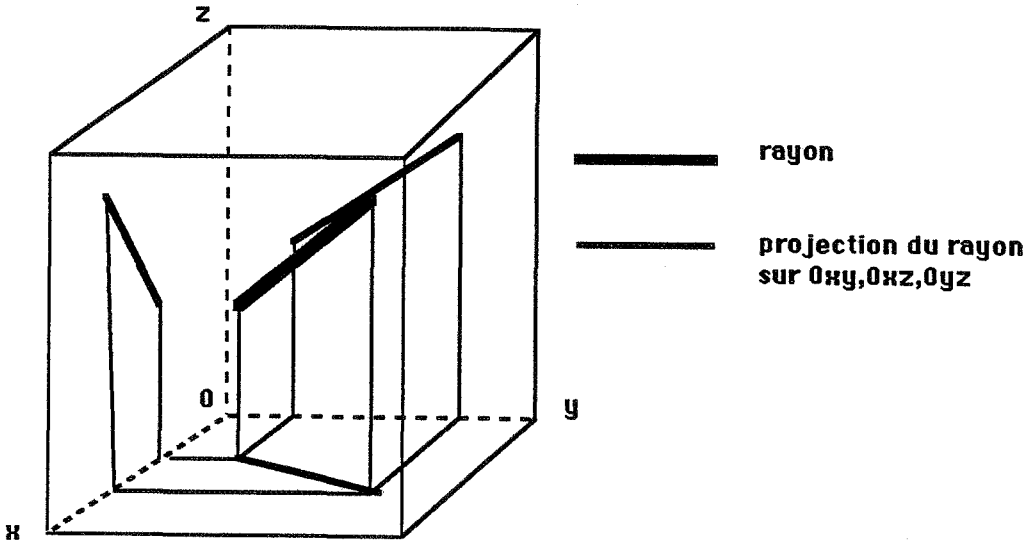


Fig. 10 Projection du rayon sur les trois quadtrees.

La structure des quadtrees que nous utilisons est composée de quatre pointeurs sur la même structure, un sous-arbre CSG et un champ qui pourra prendre les valeurs PLEIN, VIDE et PARTIEL.

- PLEIN : la boîte englobante de l'objet se trouvant dans la cellule contient les quatre frontières de la cellule,
- VIDE : la cellule ne contient aucun objet,
- PARTIEL : (non VIDE) et (non PLEIN)

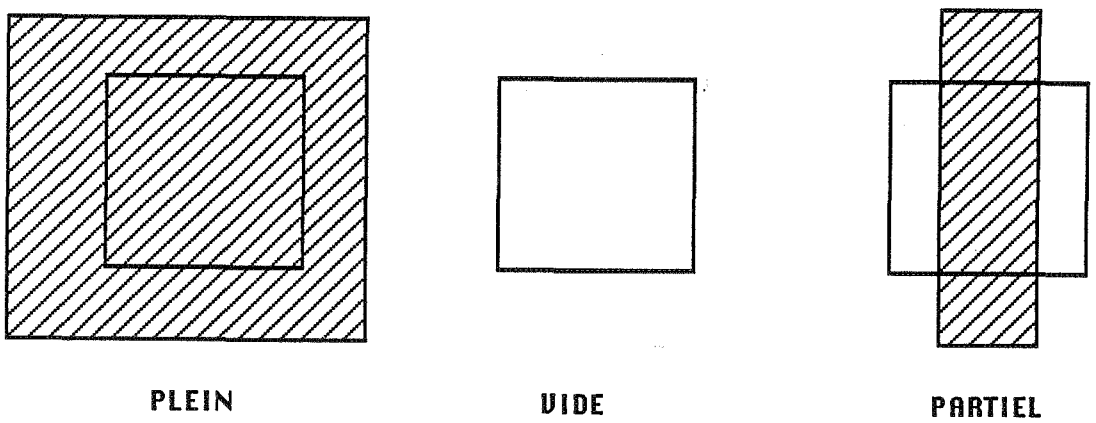


Fig. 11 Occupation d'une cellule d'un quadtree.

5.1 Création des trois quadtrees

Afin de réduire le temps de construction et de consultation des quadtrees, de nombreux tests et calculs sont faits sur des entiers. Pour passer des coordonnées (x,y,z) dans le repère de la scène à des coordonnées (X,Y,Z) dans le repère d'un quadtree on

procède de la façon suivante :

$$X = \frac{2^p (x\text{-}xmin)}{(xmax\text{-}xmin)},$$

$$Y = \frac{2^p (y\text{-}ymin)}{(ymax\text{-}ymin)},$$

$$Z = \frac{2^p (z\text{-}zmin)}{(zmax\text{-}zmin)};$$

où p est la profondeur du quadtree,
xmin,xmax,ymin,ymax,zmin,zmax les caractéristiques de la boîte englobant la scène.

L'algorithme de création d'un quadtree peut s'écrire :

```
construire_quad(quadtree,scène)
   SI la scène a un fils gauche
   ALORS
      construire_quad(quadtreeg,scène->filsg);
      construire_quad(quadtreed,scène->filsd);
      fusion(quadtree,quadtreeg,quadtreed,scène->opérateur,2p);
   SINON
      quadtree->sous_scène = scène;
   FINSI
```

La procédure *fusion* fusionne les deux quadtrees ; elle agit différemment suivant que l'on se trouve ou non à la profondeur maximale du quadtree. En effet, à la profondeur maximale, il faut fusionner les parties d'arbres se trouvant dans les deux quadtrees gauche et droit compte tenu de l'opérateur ensembliste qui les relie. Appelons *scène* le résultat de cette fusion, *scèneg* et *scèned*, les données initiales et *op*, l'opérateur ensembliste. Dans les cas suivant, il n'est pas utile de créer un nouveau nœud :

[1] *op* = UNION : si *scèneg* ou *scèned* est vide, *scène* sera respectivement *scèned* ou *scèneg*,

[2] *op* = DIFFERENCE : si *scèneg* est vide, *scène* est vide; si *scèned* est vide, *scène* sera *scèneg*,

[3] *op* = INTER : si *scèneg* ou *scèned* est vide, *scène* est vide.

L'algorithme fusion s'écrit de la manière suivante :

```
fusion( quadtree, quadtreeg,quadtreed,op,niveau,indi,indj)
{
si niveau est égal à 1
   quadtree->scène = opérateur_testant_les_scènes_vides(op,
                                          quadtreeg->scène,quadtreed->scène);
sinon {
   si quadtreeg et quadtreed ont le même niveau de subdivision {
      si selon_cas_subdivision( plein(quadtreeg), plein(quadtreed)) {
         /* les cas de subdivision seront exposés dans les tableaux suivants */
         subdiviser(quadtreeg, niveau, indi, indj);
         subdiviser(quadtreed, niveau, indi, indj);
         }
      }
      sinon
         subdiviser celui qui est le moins subdivisé;

      fusion(quadtree->fils1,quadtreeg->fils1,quadtreed->fils1,op,niveau/2,
                  indi,indj);
      fusion(quadtree->fils2,quadtreeg->fils2,quadtreed->fils2,op,niveau/2,
                  indi+niveau/2,indj);
      fusion(quadtree->fils3,quadtreeg->fils3,quadtreed->fils3,op,niveau/2,
                  indi,indj+niveau/2);
      fusion(quadtree->fils4,quadtreeg->fils4,quadtreed->fils4,op,niveau/2,
                  indi+niveau/2,indj+niveau/2);

   }
}
```

Si l'on ne se trouve pas à la profondeur maximale des quatrees, il faut subdiviser chacun des deux quadtrees en quatre sous-quadtrees. Puis, on teste si les boîtes englobantes des scènes qui se trouvent dans les quadtrees pères se projettent dans un quadtree fils, pour déterminer si l'on doit rappeler, récursivement, la procédure de fusion. Ainsi, on fusionne les fils1, fils2, fils3 et fils4 du quadtree gauche avec respectivement les fils1, fils2, fils3 et fils4 du quadtree droit.

Néanmoins, selon le remplissage des cellules des deux quadtrees gauche et droit, caractérisé par "plein", il peut s'averer inutile de subdiviser. Nous présentons tous les cas possibles dans les trois tableaux suivants, dans lesquels *pleing* et *pleind* représentent les champs "plein"des quadtrees gauche et droit (*quadtreeg* et *quadtreed*); *scèneg* et *scèned* les scènes associées aux deux quadtrees; *scène* sera le résultat mis dans quadtree.

Cas 1 : op = UNION

pleing pleind	VIDE	PLEIN	PARTIEL
VIDE	plein : VIDE scène : NIL	plein : PLEIN scène : scèneg	plein : PARTIEL quadtree ← quadtreeg
PLEIN	plein : PLEIN scène : scèned	plein : PLEIN scène : union de scèneg et scèned	plein : PARTIEL subdiviser
PARTIEL	plein : PARTIEL quadtree ← quadtreed	plein : PARTIEL subdiviser	plein : PARTIEL subdiviser

cas 2: op = DIFFERENCE

pleing pleind	VIDE	PLEIN	PARTIEL
VIDE	plein : VIDE scène : NIL	plein : PLEIN scène : scèneg	plein : PARTIEL quadtree ←quadtreeg
PLEIN	plein : VIDE scène : NIL	plein : PLEIN scène : diff. de scèneg et scèned	plein : PARTIEL subdiviser
PARTIEL	plein : VIDE scène : NIL	plein : PARTIEL subdiviser	plein : PARTIEL subdiviser

cas 3: op = INTER

pleing pleind	VIDE	PLEIN	PARTIEL
VIDE	plein : VIDE scène : NIL	plein : VIDE scène : NIL	plein : VIDE scène : NIL
PLEIN	plein : VIDE scène : NIL	plein : PLEIN scène : inter de scèneg et scèned	plein : PARTIEL subdiviser
PARTIEL	plein : VIDE scène : NIL	plein : PARTIEL subdiviser	plein : PARTIEL subdiviser

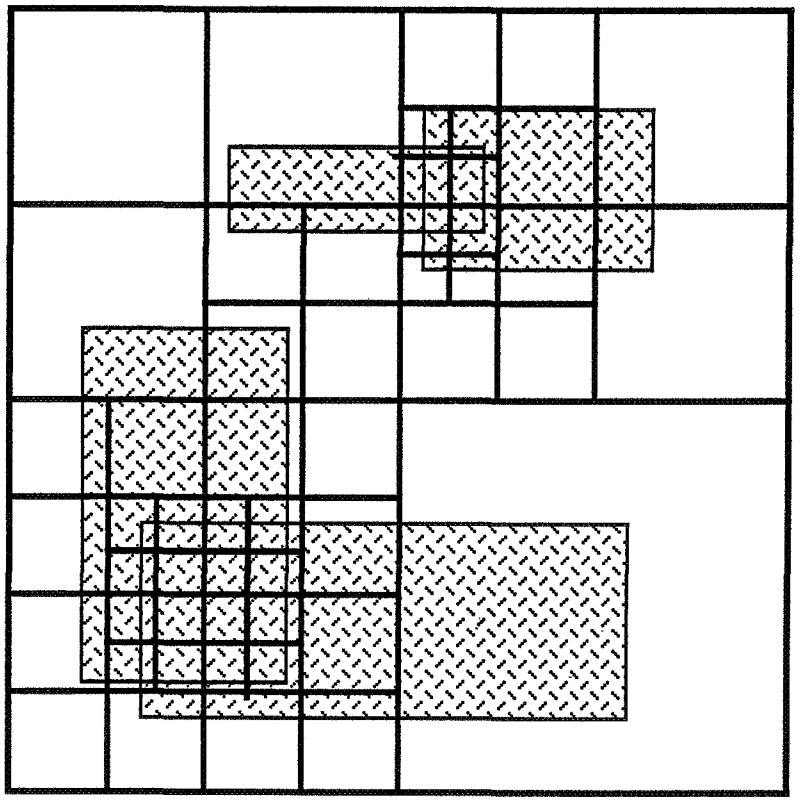


Fig. 12 Un exemple de quadtree, de profondeur 4.

5.2 Utilisation des quadtrees

Nous allons exposer les diffèrents problèmes que pose l'utilisation d'un quadtree. Tout d'abord nous présenterons l'algorithme d'intersection entre un rayon et un quadtree. Nous discuterons, ensuite, sur le moyen d'éviter de calculer plusieurs fois l'intersection avec une primitive. Ensuite nous aborderons la détermination de la cellule de l'octree de départ, puis pour un rayon donné le passage à la cellule suivante. Enfin, nous essayerons de déterminer de manière automatique la profondeur maximale du quadtree.

5.2.1 Algorithme d'intersection

L'algorithme d'intersection peut s'écrire de la manière suivante:

```
inter_quad(rayon)
    Déterminer le quadtree à utiliser;
    Calculer le point de sortie du quadtree;
    Calculer la cellule de départ où se projette le rayon;
    SI la scène,c_scène, contenue dans la cellule est non vide
        Inter(rayon,c_scène);
    FSI
    SI intersection
        Tester l'appartenance du point d'intersection à la cellule;
        SI le test est négatif
            Pas d'intersection;
        FSI
    SINON /* pas d'intersection */
        TANTQUE l'on n'atteint pas la cellule de sortie et qu'il n'y a pas d'intersection
            Chercher la cellule suivante;
            SI la scène, c_scène, contenue dans la cellule est non vide et l'intersection n'a
pas déja été calculée
                inter(rayon,c_scène);
            FSI
            SI intersection
                Tester l'appartenance du point d'intersection à la cellule;
                SI le test est négatif
                        Pas d'intersection;
                FSI
            FSI
        FTANTQUE
    FSI
```

Le problème que pose l'utilisation d'un quadtree (ou d'un octree d'ailleurs) est qu'un test d'intersection avec une primitive peut être effectué plusieurs fois pour un même rayon. La solution que nous proposons est la suivante : chaque primitive garde le numéro du dernier rayon testé (chaque rayon possède un numéro unique). Si le rayon n'intersecte pas la primitive, il est inutile de retester cette primitive dans une cellule suivante. En revanche, s'il y a intersection, il faut manœuvrer avec beaucoup plus de prudence. En effet, cette intersection peut se trouver à l'extérieur de la cellule étudiée. Dans ce cas, elle n'est pas valide, car dans la cellule suivante il peut exister une primitive non encore testée qui se trouve devant le point d'intersection trouvé. Pour résoudre ce cas, on peut soit ignorer une intersection non valide, soit stocker dans la primitive cette information pour ne pas la recalculer dans une cellule voisine. La méthode de Amantides et Woo[AMAN87] ne peut pas être utilisée puisque nous travaillons sur un arbre CSG et garder l'information

du point le plus proche n'est pas suffisante.

5.2.2 Recherche de la cellule contenant un point

Comme nous l'avons remarqué précédemment, le point de départ d'un rayon de réflexion ou de réfraction se trouve à l'intérieur de la boîte englobante de la scène. Donc l'origine du rayon se projette dans chacun des quadtrees. Pour un quadtree donné, nous considèrerons qu'un point appartient à une de ses cellules, si ses coordonnées sont strictement inférieures aux coordonnées du coin supérieur droit de la cellule et supérieures ou égales aux coordonnées du coin inférieur gauche. Tous les tests sont effectués sur les partie entières des coordonnées du point ramené dans le repère du quadtree. Les calculs sont donc faits en entier afin de réduire le temps pris par ces tests.

La recherche de la cellule contenant un point ne sert pas uniquement à déterminer la cellule de départ, mais aussi à touver la cellule suivante.

5.2.3 Recherche de la cellule suivante.

Le principe de la méthode consiste à touver un point se trouvant sur le rayon et dont on est sûr qu'il se trouve dans la cellule suivante. Connaissant ce point, il est facile de trouver la cellule suivante. L'algorithme, performant, pour trouver la cellule suivante, est basé sur l'algorithme proposé par Kunii et Wyvill [KUNI85].

On considère toujours qu'un point P appartient à la cellule dont le coin inférieur gauche est L et le coin supérieur droit H, si :

$$L[i] \leq P[i] < H[i], \text{ avec } i = (0,1)$$

Soit P_0 le point de la cellule courante, P le point de la cellule suivante, D la direction du rayon. On peut écrire l'algorithme de la manière suivante :

```
POUR i = 0 à i = 1
   SI D[i] ≥ 0     R[i] = H[i] - P0[i];
   SINON           R[i] = L[i] -   P00[i] - 1;
   FSI
FPOUR
 k = 0;
SI abs(R[0]*D[1]) > abs(R[1]*D[0])  k = 1;
FSI
POUR i = 0 à i =1
   SI i == k       P[i] = P0[i] + R[i];
   SINON           P[i] = P0[i] + (D[i]*R[k]) / D[k];
FPOUR
```

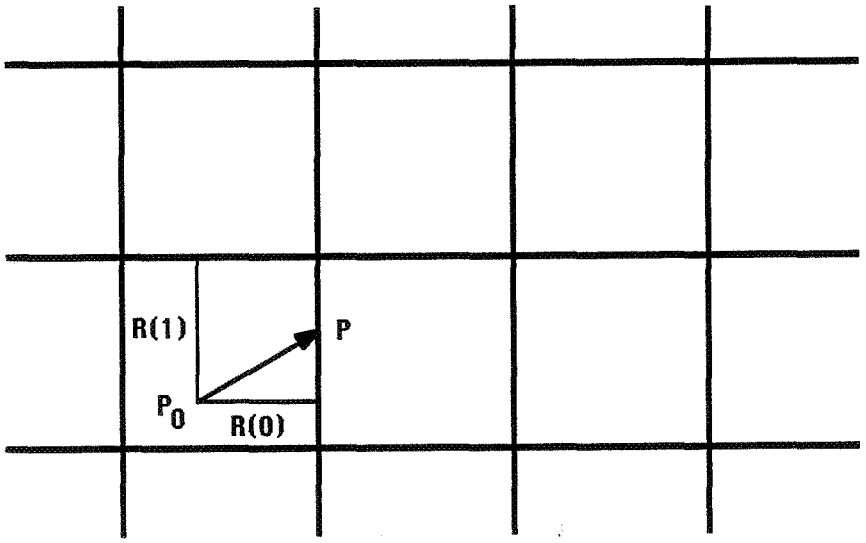


Fig 13. Recherche de la cellule suivante.

5.2.4 Détermination de la profondeur des quadtrees

Soit n le nombre de primitives dans la scène. Le but des quadtrees est de discriminer ou autrement dit séparer, si cela est possible, les diffèrentes primitives. Si les primitives sont équiréparties dans l'espace, une profondeur de l'octree de $\log\sqrt{n} = \frac{\log n}{2}$ sera nécessaire. Si ces primitives sont séparables, au plus une profondeur de logn sera utile. Il faut donc choisir une profondeur p comprise entre $\frac{\log n}{2}$ et logn. On pourrait prendre logn, mais ce n'est pas forcément optimal, et cela prend plus de place. Nous laisserons l'utilisateur libre du choix de la profondeur, mais celle-ci sera instanciée par défaut à $\frac{3\log n}{4}$, c'est à dire le milieu des bornes que nous venons de préciser. Les résultats expérimentaux valident ce choix.

6 Optimisations du calcul d'intersection entre un rayon et un arbre de construction : instanciation

Le choix du terme instanciation n'est pas fortuit. En effet, le parallèle avec les langages à objets est évident. On possède des classes de primitives : cube, sphère, cône ... . Chaque objet de la scène hérite des caractéristiques de sa classe (topologique, par exemple un ellipsoïde hérite de la procédure d'intersection d'un rayon avec une sphère unité). De plus, on instancie ses caractéristiques géométriques (sa position dans la scène, sa taille ...) et son aspect (couleur, texture, transparence ou réflexion ...). D'ailleurs, la méthodologie orientée objet a été utilisée pour construire un algorithme de tracé de rayons [YAMR87], l'implantation a été faite en C avec l'utilisation de LYMB un langage à objets développé par le centre de recherches et de développement de General Electric.

Nous n'utiliserons pas un langage à objets. Ce serait trop pénalisant au niveau du temps de calcul. Mais nous utilisons la notion d'héritage en l'étendant à ce que nous appellerons un héritage local. En effet, chaque primitive hérite dans son repère local de la position de l'œil et de la position de l'écran. Pour une vue donnée, ces paramètres ne se modifient pas. L'œil est donné par un point, l'écran par son centre et deux vecteurs qui définissent les cotés d'un pixel dans le repère local. Le repère de l'écran n'a aucune raison d'être orthonormé dans le repère local, mais le caractère affine de la matrice 4x4 associée permet de déterminer le vecteur directeur du rayon passant par le pixel (i,j) non plus dans le repère du monde, mais directement dans le repère local de la primitive devant être testée.

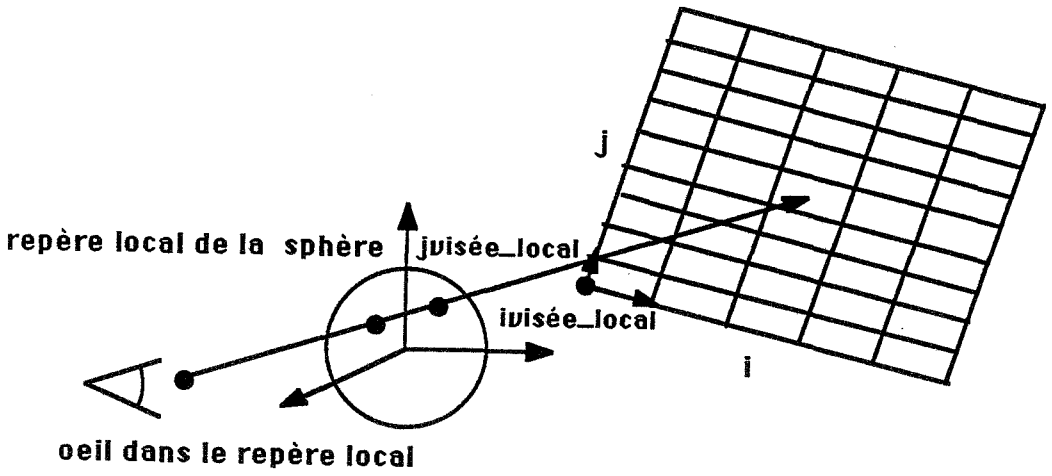


Fig. 14 Repère local d'une primitive sphère

ray_local = i*ivisée_local + j * jvisée_local + position_local;
où position_local = centre_repère_écran_local - œil_local;

Pour passer le rayon dans le repère local, il suffit, par conséquent, de 6 additions et 6 multiplications au lieu de 24 additions et 32 mutliplications en mutlipliant l'origine de l'œil et le vecteur directeur du rayon par la matrice 4x4 de passage de la scène dans le repère local. Mais, il est nécessaire d'initialiser ces instances par quatre multiplications d'un point (ou, ce qui revient au même, d'un vecteur) par la matrice de passage de la scène au repère local (matrice que l'on prend 4x4), ce qui représente 52 additions et 64 multiplications pour chaque primitive à instancier dans la scène. Malgré le coût de l'initialisation, cette optimisation est valable pour la simple raison que le nombre de primitives est largement inférieur au nombre de pixels !

Pour les rayons secondaires dirigés vers une source lumineuse, l'instance de la primitive peut aussi hériter, dans son repère local, de la position de la source lumineuse, si celle-ci est à distance finie, ou de sa direction, si elle est à l'infini (c'est à dire un point ou un vecteur). Ceci n'est évidemment plus valable pour les rayons réfléchis ou diffractés puisque leur position et leur direction est arbitraire.

La connaissance de la taille d'un pixel dans le repère local permet de connaitre le cœfficient de compression de la texture qui est (éventuellement) plaquée sur la primitive.

7 Extension

7.1 Les ambiguïtés et les imprécisions dans un modèle CSG

L'arbre de construction connaît un grand essor ces dernières années. Son principal concurrent, la représentation par frontières (ou Brep) a les défauts de ses qualités. En effet, dans une représentation par frontières, on connait explicitement la position de chaque face, côté et sommet. On peut ainsi, facilement, calculer des propriétés physiques (masse, centre de gravité ...). Cela implique, par contre, une structure de données complexe pour garder les notions d'adjacence de faces, de sommet commun à plusieurs faces ... Ce qui entraine aussi des problèmes d'imprécision et d'incohérence de la base de données. C'est un modèle polyédrique, il n'est pas très adapté au tracé de rayons, le résultat n'est pas de très bonne qualité. Travailler directement en Brep oblige l'utilisateur à décrire explicitement les positions des sommets. Bien évidemment, il est possible de passer d'une représentation par arbre de construction à une Brep. Les arbres de construction permettent de manipuler des objets de façon symbolique, ce qui limite des problèmes d'imprécision : une sphère est donnée par une équation et non pas par une liste de facettes.

Mais les problèmes de cohérence interviennent néanmoins. Nous présenterons tout d'abord un ensemble de problèmes posés par les arbres de construction, les solutions qui ont été apportées, leur limitations, puis les extensions que nous proposons.

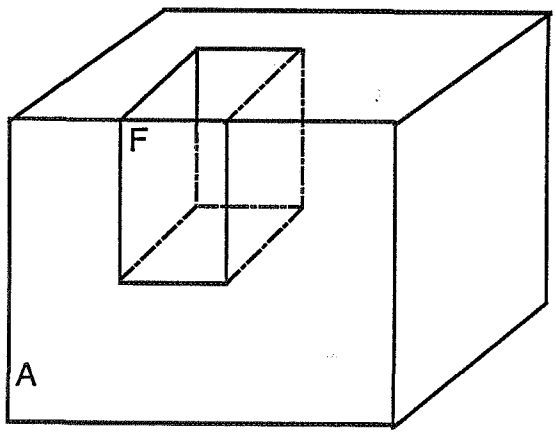


Fig. 16 Différence de deux cubes possèdant une face commune

Si deux cubes, A et B, ont une face commune F et si l'on soustrait B à A (cf Fig 14), que devient F ? La plupart des systèmes travaillant sur des arbres CSG suppriment F, ce qui impose d'abolir toute utilisation de surface et de ne s'autoriser que des volumes. Mais un utilisateur préfère en général manipuler une surface plutôt qu'un

volume dont une des dimensions vaut un epsilon (arbitrairement petit) imposé par la précision de la machine ou du logiciel [POUG87]. De plus, en tracé de rayons, les temps de calcul pour une surface, sont largement inférieur à ceux d'un volume. La solution simpliste qui consiste à supprimer ces éléments ne semble pas intêressante. Cela ne résoud par le problème voisin de la détermination de la couleur de la face F issu de l'union de A et B si A est rouge et B vert.

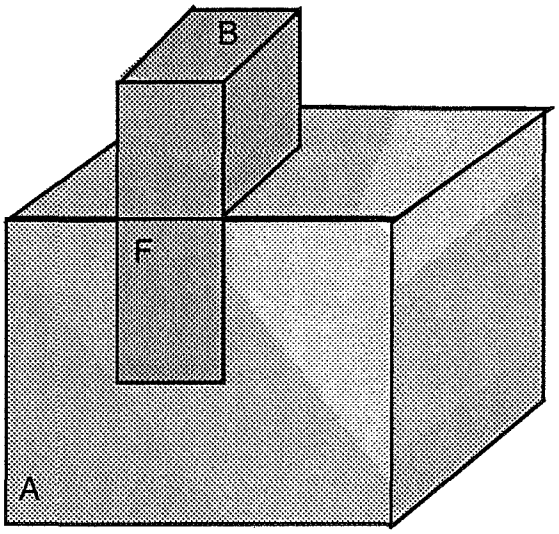


Fig. 17 Union de deux cubes possèdant une face commune

7.2 La résolution des ambiguités

Pour résoudre ces problèmes, Wyvill et Sharp, [WYVI88], suggère l'emploi d'un opérateur ensembliste à trois valeurs DEDANS, DEHORS, BORD, ce qui, avec l'ordre induit par l'arbre CSG, droit est avant gauche permet de résoudre les conflits et les imprécisions que nous venons d'exposer. L'algorithme de détermination de la couleur, après que le point d'intersection, pinter, et la primitive intersectée, la_feuille, ont été trouvés, se décompose de la manière suivante :

```
déterminer(pinter,scène,la_feuille,couleur)
/* couleur est paramètre de sortie qui ramène l'information de couleur */
   TCOULEUR  tempcouleur;
   SI  scène est une feuille
      SI scène = la_feuille       retourner(BORD);
      SINON  retourner( intérieur(scène,pinter))
      couleur = la_couleur(scène);
   SINON
      SI  scène est une union
         CHOIX (déterniner(pinter,scène->filsdroit,la_feuille,couleur))
            DEDANS:   retourner(DEDANS);
            DEHORS:   retourner(déterminer(pinter,scène->filsgauche,couleur)
            BORD:     SI déterminer(pinter,scène->filsgauche,tempcouleur) =
DEDANS
                           couleur = tempcouleur;
                           retourner(DEDANS);
                      SINON
                           retourner(BORD);
         FCHOIX
      SINON
      /* c'est une différence */
      CHOIX (déterniner(pinter,scène->filsdroit,la_feuille,couleur))
            DEDANS:   retourner(DEHORS);
            DEHORS:   retourner(déterminer(pinter,scène->filsgauche,couleur)
            BORD:     SI déterminer(pinter,scène->filsgauche,couleur) = DEDANS
                           retourner(BORD);
                      SINON
                           retourner(DEHORS);
         FCHOIX
      FSI
   FSI
```

Cette méthode a l'inconvénient de supprimer l'associativité de l'union. En effet, A union B pourra donner un résultat différent de B union A. Cela demande à l'utilisateur une gymnastique compliquée, pour construire un arbre qui soit exactement le reflet de l'ordre désiré. De plus, cette méthode demande de reparcourir la scène en testant si un point appartient à un volume. Même si l'on utilise les boîtes englobantes pour tronquer des tests avec des sous-arbres non concernés, cela rajoute un certain temps de calcul. Si ce test est fait lors du calcul du point d'intersection, on ne peut pas travailler sur des arbres réduits qui peuvent présenter un ordre différent de celui de l'arbre contenant toute la scène.

Un autre problème entrainé par l'utilisation d'abre de construction est la détermination de la couleur, par exemple, d'un trou fait par un cylindre rouge dans un cube bleu. Dans la réalité, un trou fait dans un matériau est de la couleur du matériau. Dans le cas d'un arbre de construtions, c'est un peu plus compliqué. Soient A et B deux objets, par exemple A est un cube bleu et B un cylindre rouge. Dans la plupart des systèmes utilisant des CSG, A-B sera un cube avec un trou rouge. En effet, ils travaillent sur la détermination de la surface visible et non pas sur une notion de volume. Cette solution peut être suffisante dans la plupart des cas : il suffit que le cylindre soit aussi de couleur bleu, plus généralement, que le matériau composant A et B soit le même. Dans l'algorithme précédent de détermination de la couleur, le trou sera systématiquement bleu, la possibilité de faire un trou d'une autre couleur que le matériau creusé est refusée. Remarquons, d'autre part, que ce problème ne se pose pas uniquement pour la couleur, mais aussi pour la détermination de la texture, des coefficients de réfraction ou de réfléxion, la masse ... pour deux objets occupant le même espace. De manière plus générale, ce problème se pose pour toutes les caractéristiques des matériaux utilisés. Mais la solution qui oblige à garder une cohérence des matériaux de deux objets occupant le même espace est très limitative et coûteuse en temps de calcul. Par exemple, si l'on veut mettre des poissons dans un bocal contenant de l'eau, on doit retrancher le volume en eau des poissons avant des les mettre dans ces trous. On doit faire "(A moins B) union B". Ce qui augmente considérablement la complexité, en nombre d'objets, de la scène. Wyvill et Sharp, pour contourner ce problème, introduisent un nouvel opérateur $\leftarrow\!\!\!+$ qui est une union assymétrique :

$$A \leftarrow\!\!\!+ B = (A - B) \cup B$$

Ils parviennent donc à résoudre les problèmes posés par l'union de deux objets qui occupent un même espace. Mais le problème que pose la détermination des caractéristiques du matériau de l'intersection de deux objets n'est pas résolus. Leur solution oblige l'utilisateur à lever lui même l'indétermination. On pourrait souhaiter lever cette indétermination de façon automatique par des notions naturelles, par exemple, si l'utilisateur met un objet opaque dans un objet transparent, il veut voir cet objet opaque à travers l'objet transparent. Il ne faut pas que le système s'arrête en indiquant une erreur, ou qu'il supprime l'objet opaque. Mais il faut aussi laisser l'utilisateur libre de changer ces déterminations quand les automatismes ne répondent pas à ses besoins. Pour ce faire nous utilisons trois critères, par ordre décroissant d'importance :

[1] un nombre, x, que l'utilisateur peut définir pour induire un ordre sur les objets, par défaut ce nombre vaut 0,

[2] le coefficient, k_t, de transparence de chaque primitive,

[3] le numéro, num, de chaque primitive, ce numéro est donné, de façon unique, par défaut.

Etant donné deux objets A (x_A,k_{tA},num_A) et B (x_B,k_{tB},num_B), A sera prédominant devant B si A est avant B dans l'ordre lexicographique i.e.:

```
A >> B
   SI xA > yB
   SINON
      SI xA = yB
         SI ktB < ktA
         SINON
            SI numA > numB
```

Cet ordre permet de choisir la primitive qui donnera le type de matériau du volume commun. Mais, nous voulons donner plus de liberté au système. Le choix, quand deux volumes, occupent le même espace doit pouvoir découler des propriétés suivantes :

[1] propriétés surfaciques : les caractéristiques de la surface rencontrée sont déterminées par la primitive qui contient cette surface. On en déduit la couleur, la texture, la réflexion... C'est utile pour déboguer une base de données, ou pour une utilisation en CAO. On notera cette opération entre deux objets A et B, A s B. Par contre, la transparence est une propriété du matériau et non plus de la surface, on n'autorisera pas un volume à posséder plusieurs coefficients de transparence.

[2] propriétés volumiques : pour conserver une cohérence physique les caractéristiques d'un volume ne doivent par dépendre de la surface rencontrée mais être uniques pour le volume. Quel sera le matériau de la matière commune à deux primitives A et B ? Si A >> B ce sera la matériau de A, si B >> A se sera le matériau de B, sinon on fera un mélange des propriétes de A et B : A ~ B ([CARE85]).

On veut une situation par défaut qui permettra de déboguer la base de données tout en gardant une cohérence physique pour les objets transparents (un objet opaque sera visible à l'intérieur d'un objet transparent). Mais on veut laisser l'utilisateur libre de spécifier ses propres lois, qui seront liées à chaque objet et non pas globales pour toute la scène. Pour cela, on étend la signification de x dans l'ordre << :

$\varepsilon x,y$ avec $\varepsilon = (-1, 0, 1)$, x entier naturel et 0.y $\in [0,1]$. Par exemple -5,0654.

Comparons, maintenant, les priorités de deux objets A ($\varepsilon_A x_A, y_A$) et B ($\varepsilon_B x_B, y_B$).

```
SI xA > xB    ALORS  A >> B
SINON
   SI xB > xA    ALORS  B >> A
   SINON
      SI ((εA ≥ 0) ou (ε B ≥ 0)) et xA = xB
         SI 0 ≤ ktA < ktB  ALORS  A >> B                    (1)
         SINON
            SI 0 ≤ ktB < ktA  ALORS  B >> A                 (2)
            SINON
                  SI 0 ≠ ktA ≠ ktB ALORS
                        SI numA > numB ALORS  A >> B        (3)
                        SINON  B >> A
                  SINON  /* 0 = ktA = ktB */   A  s  B
      SINON /* (εA = -1) et (ε B= -1)) et x A= xB */     A ~ B
```

Par défaut, (εx,y) prendra la valeur 0. Ce qui nous limite donc bien aux cas (1), (2) et (3). Les problèmes d'imprécision sont réglés par :

```
SI    ((distance(Point entrée(A),Point entrée(B)) < EPS_PREC) OU
       (distance(Point sortie(A),Point sortie(B))   < EPS_PREC) ) ALORS
      matière = eval _ordre ( A, B);

eval_ordre( A, B)
   SI    A >> B   ALORS retourner matière(A);
   SI    B >> A   ALORS retourner matière(B);
   SI    A ~ B    ALORS retourner matière(A) ~ matière(B)
   SI    A  s  B  ET num_prim(A) >  num_prim(A) ALORS retourner matière(A);
   SI    A  s  B  ET num_prim(B) >  num_prim(A) ALORS retourner matière(B);
```

En revanche, un objet surfacique sera considéré comme possédant une profondeur EPS_EP = 2 EPS_PREC. Ce qui permet de traiter un élément surfacique de la même manière qu'un élément volumique. Cette épaisseur est définie automatiquement, l'utilisateur n'a pas à s'en préoccuper.

Cette optique permet de concilier les différents modèles, en gardant une cohérence physique et en économisant du temps de calcul. Nous allons maintenant étendre la notion d'arbre CSG à de nouvelles primitives qui nous permettrons d'autres optimisations.

7.3 Les primitives creuses

L'oeil ne peut se trouver à l'intérieur d'une primitive volumique. Pour que l'œil puisse être dans une pièce, par exemple, il faut prendre un cube élémentaire C_E et lui retrancher un cube plus petit (moyennant quelques changements d'échelle). La matière, étant ainsi creusée, l'œil peut se trouver à l'intérieur du cube C_E. Si l'œil est à l'extérieur de la pièce, on devra faire une opération ensembliste bien que seul l'extérieur soit visible. Une autre méthode consisterait à créer quatre murs, un plafond et un plancher puis à les unir pour former la pièce. D'une part ce n'est pas vraiment dans l'esprit de la méthodologie CSG, d'autre part, cette méthode, comme la précédente, augmente la compléxité de la scène et forcément les temps de calculs. Compte tenu des optimisations que nous avons faites, il est fort probable que les temps de calcul de la première méthode soient supérieurs à ceux de la seconde malgré un nombre d'objet largement supérieur pour la seconde. En effet, dans le cas d'unions, c'est le minimum des points d'entrée qui est recherché tandis que pour des différences c'est un tri sur les points d'entrée et de sortie des primitives. De plus, les objets manufacturés sont souvent issus de la différence d'une forme moins son homothétique : vase, verre, bouteille, paille, couloir, pièce, tuyau, poubelle, casserole ... Nous allons définir des primitives creuses de la manière suivante :

soit p une primitive volumique de type X, la primitive creuse p_c de type X_c sera égale à
p_c = p - affinité(p,x,y,z,étoilé(p)) avec x,y,z dans] 0, 1 [et étoilé(p) est un point étoilé de p*.

Le fait que l'affinité aient des coefficients strictement inférieur à 1 et qu'elle soit centrée sur un point étoilé de p entraîne la particularité suivante : si l'œil est à l'extérieur, seule la paroi extérieure est visible, si l'œil est à l'intérieur, seule la paroi intérieure. Il n'est donc plus utile de faire une opération ensembliste pour visualiser p_c, il suffit de détecter si l'œil est à l'intérieur ou à l'extérieur pour une vue donnée. Le premier point d'intersection suffit . Nous n'utiliserons l'opérateur ensembliste que si cette primitive intervient à l'intérieur d'une différence, intersection ou un rayon de transparence, car nous avons alors besoin de connaître l'épaisseur de la primitive. Les méthodes d'optimisations que nous avons proposées dans ce chapitre s'adaptent très bien (cf Fig. 17).

---

* un point M est étoilé pour p, si pour tout point N de p, le segment MN est entièrement inclus dans p. On peut choisir, par exemple, le centre de symétrie de la primitive.

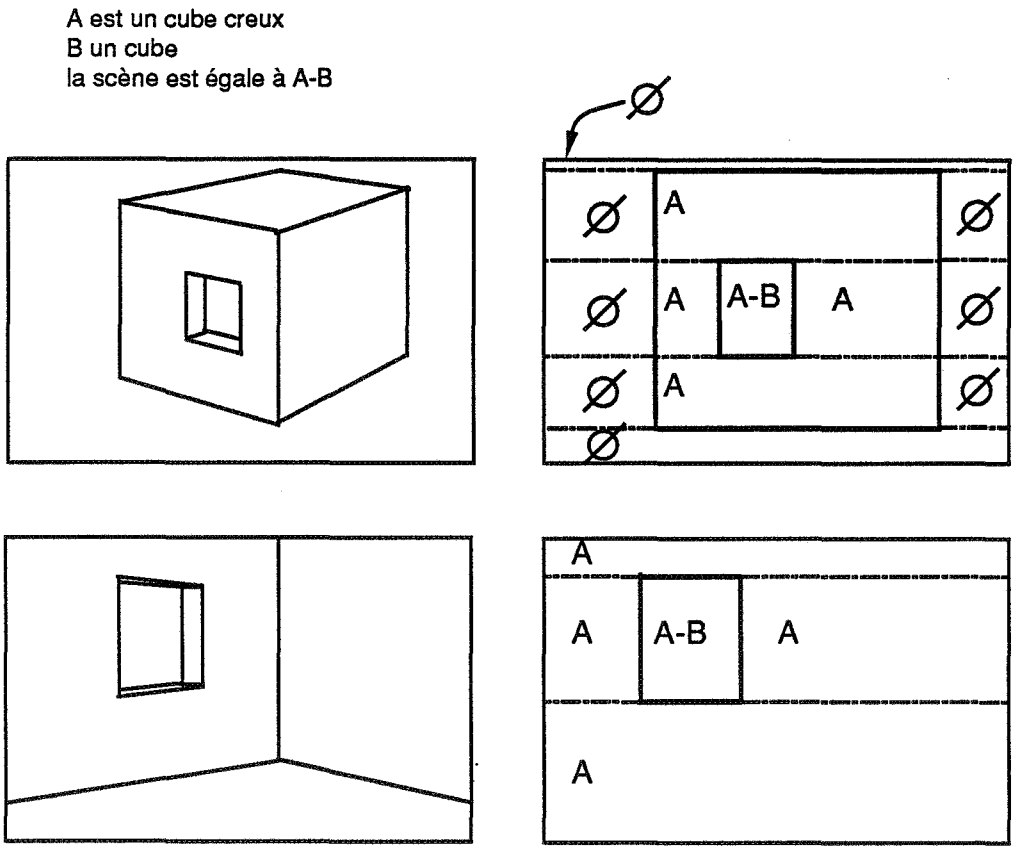


Fig. 19 Optimisations pour un objet creux

Ces primitives creuses permetttent aussi de définir, naturellement, des niveaux d'imbrication dans la scène. Imaginons une maison comportant un mobilier et des éléments complexes (escaliers ...). Vus de l'extérieur, ces objets ne seront visibles qu'à travers les fenêtres ; il est inutile de les garder si leur projection se trouve à l'extérieur de la projection de la fenêtre sur l'écran. Nous pouvons ainsi limiter la complexité de la scène et par conséquent, les temps de calcul. Lors de la lecture de la scène, nous pouvons définir, de façon automatique, une arborescence d'inclusion en construisant, tout d'abord, un graphe d'inclusion des différentes primitives creuses, puis on teste si la boîte englobante de chaque sous-arborescence complète est incluse dans l'une des primitives creuses du graphe.

Nous avons, ainsi, proposé plusieurs optimisations pour l'algorithme du tracé de rayons, les gains obtenus sont regroupés en annexe. Nous allons maintenant exposer un algorithme d'antialiassage du tracé de rayons compatible avec ces optimisations et permettant d'obtenir une image antialiasée en des temps acceptables.

Chapitre 3

Antialiassage du tracé de rayons

1. Introduction

Comme les autres algorithmes de visualisation, le tracé de rayons produit des images dont la qualité est dépréciée par l'aliassage. Mais un utilisateur peut admettre des marches d'escaliers sur les bords d'un objet créé par un logiciel de C.A.O[1], car ce qui lui importe ce n'est pas tant l'aspect d'une image que sa sémantique. En revanche, celui qui attend parfois des heures une image supposée être réaliste, admettra mal des défauts trop visibles amenés par l'aliassage. Il est donc très important de corriger l'aliassage dans le tracé de rayons, mais sans que les temps de calculs déjà trop longs soient encore multipliés.

Nous présenterons tout d'abord quelques rappels sur la théorie de l'aliassage, puis nous présenterons les différentes méthodes proposées pour résoudre ce problème pour le tracé de rayons. Enfin, nous présenterons notre méthode [ARGE88].

2. Aliassage et antialiassage

Pour bien comprendre d'où provient l'aliassage sur un écran, il convient de revenir à la théorie du signal. Le principe de la reconstrution d'un signal analogique à partir d'un signal numérique est exposé dans le théorème de Shannon :

> un signal analogique à largeur de bande finie peut être reconstitué fidèlement à partir d'une suite complète de ses échantilllons si la fréquence d'échantillonnage est au moins égale à deux fois la fréquence maximale du signal analogique.

Soit s(t) un signal analogique. Pour déterminer sa fréquence maximale, on utilise la transformée de Fourier, F(f), de s(t), définie par :

$$F(f) = \int_{-\infty}^{+\infty} s(t)\, e^{-2\pi i t f}\, dt \ .$$

F est à support borné, ie $\{f, F(f) \neq 0\}$ est un ensemble borné. La plus grande des valeurs de f tel que F(f) soit non nul, est la fréquence maximale de s(t) :

$$f_{max} = \sup_{f \in \Re} (F(f) \neq 0).$$

Il suffit donc d'échantillonner s(t) avec une fréquence supérieure à $2f_{max}$. Si la fréquence d'échantillonnage est inférieure à $2f_{max}$, apparaissent des artefacts que l'on appelle de façon générale aliassage.

---

[1] Conception Assitée par Ordinateur

Dans le cas d'une image, qui peut être assimilée à un signal bidimensionnel, les hautes fréquences correspondent aux détails. En effet, les hautes fréquences sont situées dans les zones présentant de brusques changements d'intensité, les contours des objets par exemple. Les zones de faible gradient peuvent être considérées comme des textures, avec le sens qu'on leur donne en traitement d'image : une notion de matière (aspect rugueux d'un mur crépis ...). On peut, en gros, envisager deux types de traitement de l'aliassage :

1 un pré-traitement : on augmente la fréquence d'échantillonnage.

2 un post-traitement : on diminue l'aliassage en tronquant les hautes fréquences de l'image, ce qui entraîne un léger flou. Cela peut se faire en passant un filtre passe-bas sur l'image, qui fera une moyenne des pixels et donc diminuera les brusques sauts d'intensité (et par conséquent les hautes fréquences).

Pour des contours la deuxième méthode n'est pas satisfaisante, car on perd de la précision et de la qualité. Il est donc préférable d'utiliser la première méthode. Par contre, pour une "texture" qui posséde en général moins d'informations précises de contours, la deuxième méthode est plus intéressante, car moins coûteuse en temps de calcul. En synthèse d'image, les textures sont souvent représentées sous forme de sommes de cœfficients de Fourier, il est facile de tronquer les fréquences des textures qui dépassent la fréquence d'échantillonnage de l'écran. Pour des textures tabulées, une méthode du type Williams [WILL83], Crow [CROW84] ou Ghazanfarpour [GHAZ85] peut être utilisée. Ce dernier traitement étant fait dans une phase de précalcul, il suffit ensuite de consulter une table. C'est donc une méthode rapide mais qui demande une certaine capacité de stockage. Nous présenterons dans le chapitre 5, une méthode qui tente de limiter la taille mémoire nécessaire sans pour autant trop augmenter les temps de calcul.

3. Méthodes d'antialiassage pour le tracé de rayons

Amanatides, [AMAN84], a introduit une méthode d'antialiassage qui consiste à étendre la définition d'un rayon à un cône en ajoutant l'information d'angle solide et d'origine virtuelle. Mais, les procédures d'intersection entre un "rayon" et un objet deviennent bien plus complexes. D'ailleurs, les seules procédures utilisées sont celles de la sphère, du plan, et du polygone planaire. Bien que les effets produits soient très beaux, l'image finale ne comporte que peu d'objets ([KIRK86]). Puisque l'aliassage est le produit d'un sous-échantillonnage durant la visualisation, une méthode couramment employée pour résoudre ce problème, sans perte de qualité, consiste à sur-échantillonner l'image. Cook présente, dans [COOK84], le tracé de rayons distribué[2]. Cette méthode permet d'obtenir beaucoup d'effets : la profondeur de champ, les pénombres, la transluscidité, les miroirs non parfaits, l'antialiassage d'une image y compris dans le domaine temporel. Mais cette méthode demande de sur-échantilloner pour chacun des

[2] cf chapitre 1 paragraphe 5.1.3

effets. Dippé [DIPP85], Lee [LEE 85], Mitchell [MITC87], Purgathofer [PURG86], ont combiné cette méthode avec des techniques statistiques pour choisir le motif ou la fréquence d'échantillonnage. Mais sur-échantillonner est très coûteux ; si l'on travaille sur une image de taille n^2 pixels (par exemple n = 512), et s'il l'on divise chaque pixel en p (p = 9) sous-pixels, on trouve n^2 p s ($=2^{21}$s) rayons à calculer, où s est le nombre moyen de rayons par pixel. Notons m le nombre de pixels présentant réellement de l'aliassage. Seuls $(m(p-1) + n^2)$s rayons doivent vraiment être calculés. La méthode que nous proposons diminuera le nombre de rayons à traiter de n^2 à m, ce qui est très intéressant lorsque m est largement inférieur à n^2. Comme Rubin et Whitted [RUBI80] l'ont remarqué, un algorithme de tracé de rayons passe la plupart de son temps à faire des calculs d'intersection entre un rayon et un objet, ce qui est très coûteux. Donc, tout sur-échantillonnage systématique prendra beaucoup de temps, on peut estimer le calcul d'une image neuf fois plus grande et la réduction par un filtrage à neuf fois le temps normal. Nous sommes donc assurés d'économiser beaucoup de temps en détectant les pixels qui nécessitent réellement un traitement. Whitted a introduit une méthode de détection qui cherche les brusques écarts d'intensité entre pixels voisins, car l'aliassage est plus visible dans ces zones [WHIT80]. Roth, dans [ROTH82], teste seulement la différence entre deux pointeurs sur les surfaces visibles, pour une ligne de balayage. Si la surface visible au pixel (i,j) est différente de celle vue au pixel (i,j+1), alors un rayon est envoyé entre les deux pixels.

De plus, la visualisation de scènes complexes nécessite une grande quantité de calcul. C'est pourquoi nous proposons un algorithme d'antialiassage local et adaptatif (en anglais LAO pour Local Adaptive Oversampling) du tracé de rayons. Nous qualifions notre algorithme d'adaptatif, car, comme ceux de Whitted et Roth, il sur-échantillonne uniquement les pixels présentant un problème. L'adjectif local se refère au fait que tout surcoût de calcul dû au sur-échantillonnage d'un pixel est minimisé par la restriction de la scène aux objets vraiment visibles en ce pixel. Pour chaque pixel sur-échantillonné, une sous-scène pour les rayons primaires est construite, une autre pour les rayons vers une source lumineuse, et une autre pour les rayons de réflexion, réfraction... De plus, la place mémoire nécéssaire n'est pas excessive, car nous travaillons sur une fenêtre de l'écran pour déterminer si un pixel doit être échantillonné.

3.1. Arbre de construction

Comme cela l'a été exposé dans le chapitre précédent, nous travaillons sur un arbre de construction. Mais, pour les besoins de notre méthode d'antialiassage à chaque surface C^1 de chaque primitive est associé un nombre. Par exemple, les sphères n'ont qu'une surface C^1, les cubes en ont six, les cylindres trois, les cônes deux etc... . En effet, entre deux ou plus C^1 surfaces avec des numéros différents, il y a un risque d'aliassage.

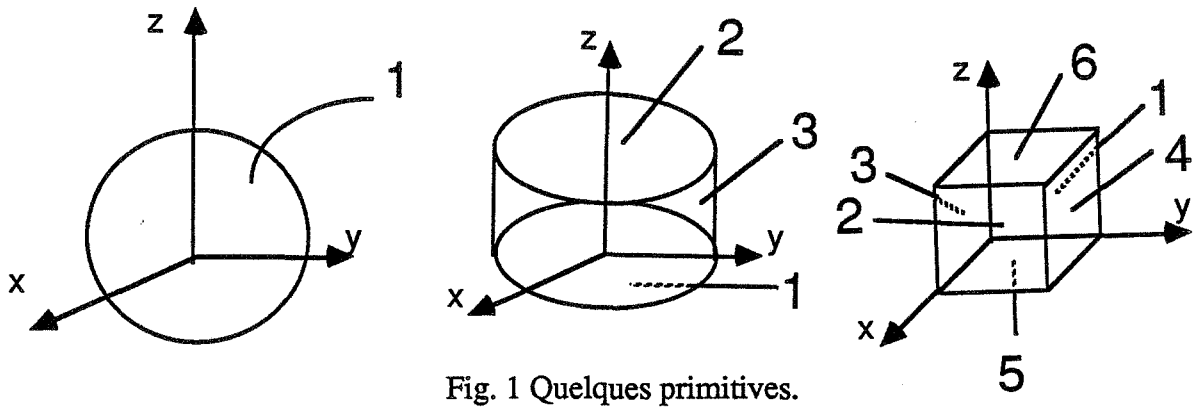


Fig. 1 Quelques primitives.

3.2. L'algorithme d'antialiassage

La procédure d'intersection entre un rayon et une primitive ne retourne pas seulement les résultats habituels : les points d'intersection (entrée, sortie), les normales en ces points, la couleur, etc... ; mais aussi le numéro de la surface associée, un drapeau indiquant si la primitive est plus petite, sur l'écran, qu'un pixel. Les petits objets, pour les rayons primaires, sont déterminés dans une phase de précalcul. Nous testons, uniquement, si la projection de leur boîte englobante, sur l'écran, fait plus ou moins d'un pixel d'épaisseur. Pour les rayons secondaires, les choses sont un peu plus compliquées. En effet, un objet peut être grand pour les rayons primaires alors que son reflet dans un autre objet est petit. Cette notion de petit objet dépend non seulement de la position de l'œil et de l'objet vu mais aussi de toutes les interactions (réflexion-réfraction-ombre) avec les autres objets de la scène. Pour résoudre ce problème, Whitted [WHIT80], a proposé l'utilisation de sphères englobantes plus grandes pour les petits objets. Ce qui, hélas, ne résout pas le problème car, pour des rayons secondaires il est difficile de déterminer quels sont les objets petits. Pour résoudre ce problème, nous choisissons d'utiliser une approximation, en créant un œil et un écran "virtuels". La position de l'œil "virtuel" est définie par le rayon incident et la distance parcourue par la lumière entre l'œil réel et le point d'intersection. Bien évidemment, cela suppose que les surfaces rencontrées soient planes ou de faible courbure.

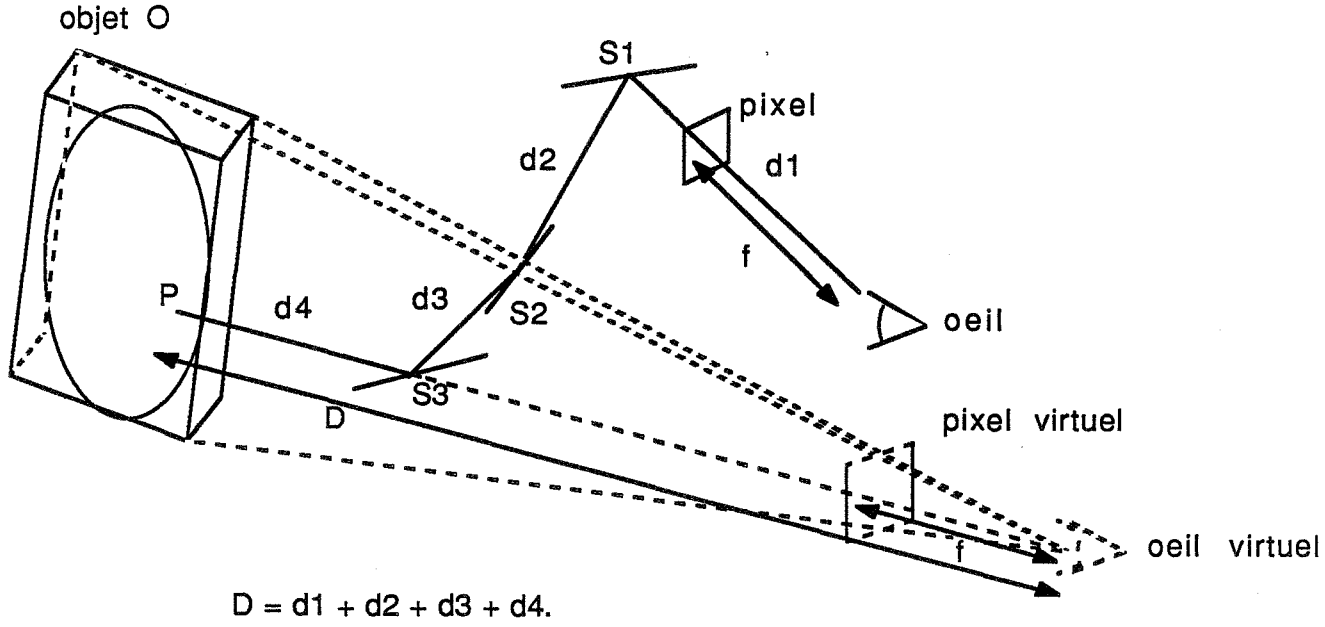


Fig. 2 Définition d'un œil virtuel pour détecter les petits objets

La taille d'un objet est calculée par rapport à la taille d'un pixel virtuel. Dans l'exemple présenté figure 2, l'objet O est petit pour le rayon incident. Mais il faut que le rayon teste l'intersection avec cet objet pour pouvoir détecter sa taille relative à un pixel. En effet, il est vraisemblable, compte tenu des volumes englobants associés à chaque nœud de l'arbre, que le rayon, ne trouvant pas d'intersection avec ce volume englobant, écarte tout simplement la possibilité d'une intersection avec les objets contenus dans le sous-arbre. Il faut vérifier après chaque test négatif entre un rayon et une boîte englobante B, si B se projette sur le rectangle associé à un pixel. En effet, si un petit objet PO est contenu dans le sous-arbre associé à B, PO se projette sur le rectangle, ainsi que sa boîte englobante et toutes les boîtes contenant PO jusqu'à B . Par conséquent, il suffit d'éliminer un sous-arbre uniquement si le volume englobant n'est pas intersecté et s'il ne se projette pas sur le pixel virtuel.

La procédure d'intersection entre un rayon et un arbre de construction retourne, en plus, le numéro de la primitive dans la scène, la liste des pointeurs sur des objets "simples" intersectés, pour chaque source lumineuse : un drapeau indiquant la présence d'ombre ou non et une liste des objets produisant de l'ombre, puis, une liste de toutes ces informations pour les rayons de réflexion et de réfraction.

Par pointeur sur un objet "simple", nous entendons un pointeur sur le premier nœud dont les ancêtres sont uniquement des unions, sans aucune intersection ni différence. Si tous les nœuds ancêtres de la primtive sont des unions, nous pouvons garder seulement un pointeur sur cette primitive. Mais, s'il existe au moins un opérateur différence (ou intersection), il n'y a pas de moyen de connaître exactement la quantité de "matière", sans garder toute l'information nécessaire à l'opérateur (cf figure suivante).

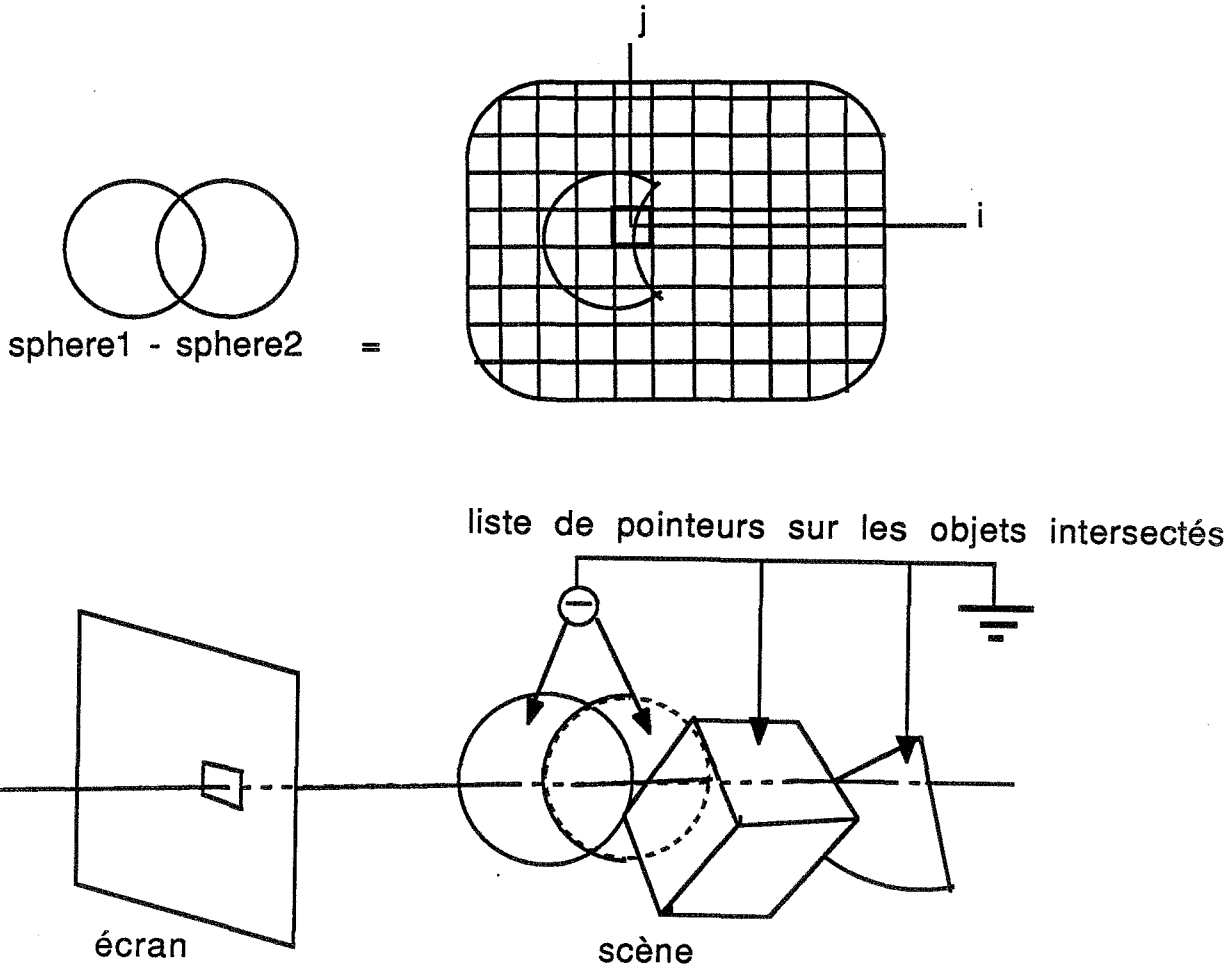


Fig. 3 Liste d'objets "simples"

Nous rappelons toutes les informations que nous devons garder :

- le numéro de la primitive intersectée,
- le numéro de la surface C^1 intersectée,
- La liste des pointeurs sur les objets simples intersectés,
- un drapeau qui indique si la primitive n'est pas éclairée par une source lumineuse,
- la liste des pointeurs sur les objets simples faisant de l'ombre pour une source lumineuse,
- la liste de ces informations pour les rayons de réflexion,
- la liste de ces informations pour les rayons de réfraction.

Nous garderons, pour chacun des pixels de trois lignes adjacentes, cette

information. Sur la ligne i, nous calculerons les informations concernant la ligne i+1. Puis, nous travaillons sur une fenêtre 3x3, sur ces trois lignes, pour réévaluer la couleur des pixels sur la ligne i en sur-échantillonnant si nécessaire.

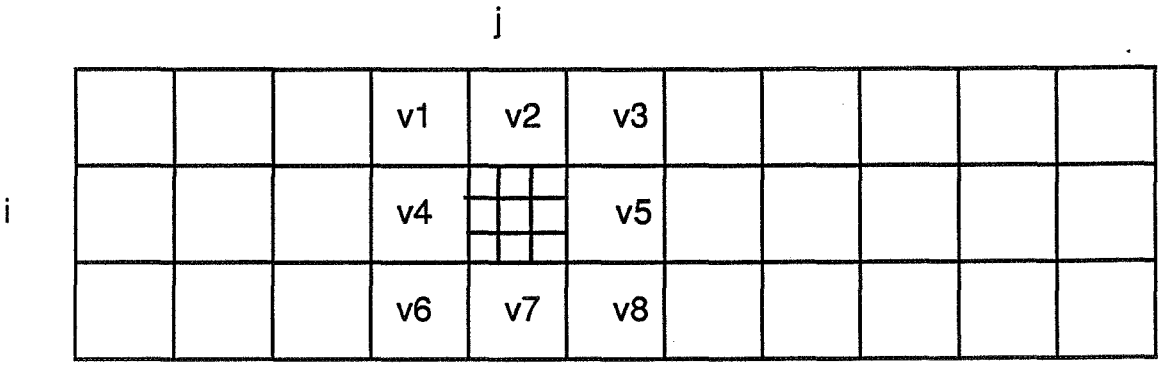


Fig. 4 Fenêtre sur l'écran.

Nous proposons, ci-dessous, le schéma principal de l'algorithme d'antialiassage :

```
principal(scène)
/*****************/
{
Calculer les deux premières lignes et garder l'information récoltée dans deux  tableaux
Info1, Info2;
POUR chaque ligne i,  i supérieur ou égal à 2 et inférieur à MAX_LIGNES
    {
    Calculer la ligne (i+1) et garder l'information dans le tableau Info3;
    POUR chaque rayon (i,j),  j supérieur ou égal à 2 et inférieur à MAX_COLONNES
        antialiasser(       Info1[j-1], Info1[j], Info1[j+1], Info2[j-1], Info2[j], Info2[j+1],
                            Info3[j-1], Info3[j], Info3[j+1]);
     Mettre Info2 dans Info1, puis Info3 dans Info2;
    }
}
```

```
antialiasser(Info_pixel1, Info_pixel2, ..., Info_pixel9)
/*********************************************************/

{
à_antialiasser = FAUX;
test_aliassage(à_antialiasser,Info_pixel1, Info_pixel2, ..., Info_pixel9);
SI (à_antialiasser est VRAI)
    {
    création d'une sous-scène pour les rayons primaires avec l'union de tous les
       objets intersectés et de tous les petits objets testés qui se trouvent dans
       Info_pixel1, Info_pixel2, ..., Info_pixel9;
    création de sous-scènes pour les rayons vers les sources lumineuses;
    création d'une sous_scène pour les rayons de réflexion;
    création d'une sous_scène pour les rayons de réfraction;
    appel d'une procédure qui va, récursivement, sur-échantillonner (i,j) en
       9 sous-pixels, calculer 8 intersections entre les sous-rayons et les
       sous-scènes et utiliser les mêmes tests que la procédure
       test_aliassage() mais sans recréer de nouvelles sous-scènes
       (ce serait une perte de temps);
    la couleur du pixel (i,j) est la moyenne des valeurs des neufs sous-pixels[3];
    }
SINON
    Il n'y a rien à faire, puisque la couleur du pixel (i,j) a déjà été calculée;
}


test_aliassage(à_antialiasser,Info_pixel1, ..., Info_pixel9)
/*********************************************************************/
/* Pour déterminer la couleur finale du pixel (i,j) on compare les 9 valeurs
   (les valeurs des 8 voisins et celle de (i,j) */

{
SI (Il y a au moins une intersection ou un petit objet testé)
   SI     (tous les neuf pixels ont le même numéro de primitive)
          ET (le même numéro de surface) ET (sont éclairés par les mêmes sources
          lumineuses) ET     (aucun petit objet n'est testé)
      SI (il n'y a ni réflexion ni réfraction)
          à_antialiasser = FAUX;
      SINON
          {
          test_aliassage (à_antialiasser,Info_pixel1->liste_réflexion,...,
                          Info_pixel9->liste_réflexion);
          SI (à_antialiasser est FAUX)
              test_aliassage     (à_antialiasser,Info_pixel1->liste_réfraction,..,
                                  Info_pixel9->liste_réfraction);
          }
   SINON
      à_antialiasser = VRAI;
SINON
   à_antialiasser = FAUX;
}
```

Bien évidemment, l'antialiassage est bien plus compliqué quand il apparait des

[3] on peut utiliser des filtres un peu plus sophistiqués q'une moyenne

transparences et des réflexions, car cela entraîne beaucoup d'appels récursifs de la procédure test_aliassage. Cela peut être amélioré en ne gardant que les feuilles de l'arbre de réflexion/réfraction et le cheminom associé à chaque feuille.

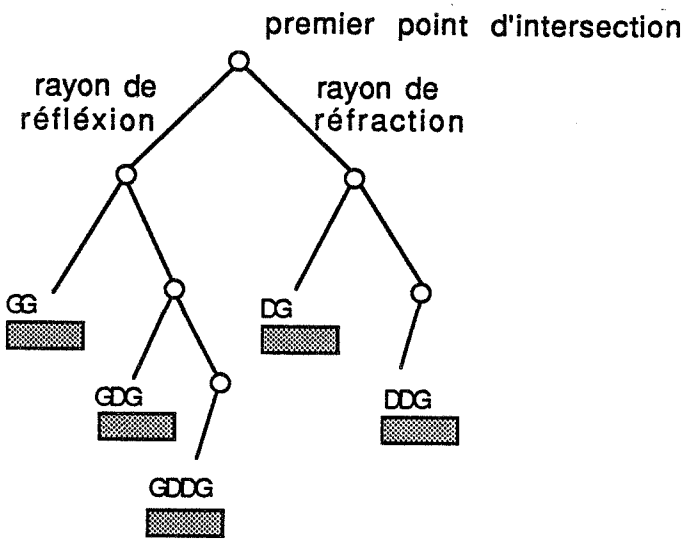


Fig. 5 arbre de réflexion/réfraction.

Si les neuf pixels n'ont pas le même nombre de feuilles et le même cheminom pour chaque feuille, de l'aliassage est détecté. Sinon, nous devons appeler, pour chaque feuille, la procédure test_aliassage. Cette simplification n'est qu'une approximation, mais les cas à problèmes sont rares.

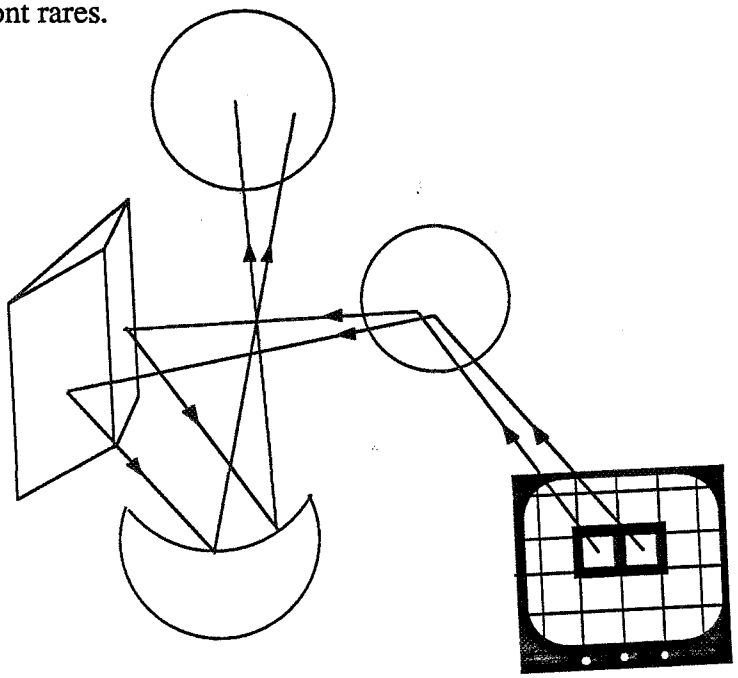


Fig. 6 cas d'une mauvaise approximation

4. Résultats et conclusion

A l'encontre de la méthode de Whitted, nous ne calculons pas la différence d'intensité entre deux pixels ; comme on a que des tests entre différentes valeurs, on gagne du temps. De plus, nous ne nous basons pas sur une décision, liée à l'œil humain, pour déterminer, pour une image donnée, quels sont les pixels à sur-échantilonner. En effet, un seuil, qui serait adéquat pour une image donnée, peut être trop faible pour une autre image. Bien entendu, dans notre méthode, il est possible que l'on sur-échantillonne dans une zone où l'œil ne verra aucun aliassage. Mais, comme le montre le tableau suivant, le nombre de pixels sur-échantillonnés est légèrement plus faible que dans la méthode de Whitted. Pour cette dernière, les composantes couleurs étant comprises entre 0 et 255, le seuil choisi est de 16. Un seuil plus élevé laissait encore apparaître des défauts d'aliassage. Nous avons, en effet, choisi ce seuil maximal pour que l'effet visuel soit le même pour les deux méthodes.

De plus, notre méthode n'antialiasse pas les pixels texturés. En réalité, le sur-échantillonnage n'est pas très bon pour les textures du type rayé ou écossais, dans les zones de haute compression. En effet, neuf pixels voisins peuvent avoir une même couleur tandis que leurs sous-pixels sont de couleurs différentes. Et, par conséquent, les problèmes de moiré ne sont pas résolus.

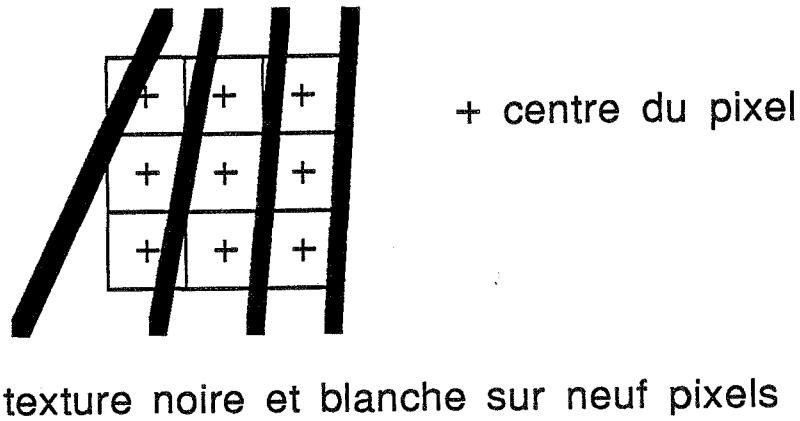


Fig. 7 problèmes de moiré non résolus.

Pour des textures définies par des fonctions mathématiques, l'antialiassage peut être fait, simplement, en testant leurs fréquences, et en filtrant celles qui dépassent la moitié de la fréquence d'échantillonnage de l'écran [NORT82]. Des textures stockées dans des tables, peuvent être antialiassées avec des méthodes similaires à celles proposées par Williams([WILL83]), Crow([CROW84]), Ghazanfarpour([GHAZ85]).

image numéro:	1		3
méthodes:	sans traitement	antialiassage utilisant la méthode de Whitted	antialiassage utilisant l'algorithme LAO
nombre de pixels:	327 680	327 680	327 680
nombre de rayons primaires intersectant des objets:	39 475	183 836	76 652
nombre total de rayons primaires:	54 353	264 055	214 725
nombre de rayons d'ombre intersectant des objets:	5 053	23 932	24 053
nombre total de rayons d'ombre:	5 271	25 194	24 413
nombre de pixels sur-échantillonnés:	0	23 904	17 824
nombre de primitives:	43	43	43
temps relatif de calcul	1	3,45	1,33

Notre algorithme semble être une solution efficace au problème de l'aliassage du tracé de rayons. Il a été développé dans un environement de modélisation basé sur les arbres de construction. Mais il peut être facilement étendu à une autre modélisation, il suffit que les objets et les faces C^1 soient déterminables et discriminables.

Chapitre 4

Couleurs

1 Introduction

La couleur entre en grande part dans la qualité et le réalisme d'une image. Tout comme l'aliassage, l'algorithme du tracé de rayons ne supporte pas une mauvaise fidélité des couleurs. En effet, il est inutile d'espérer obtenir un bon résultat d'une image où l'aliassage est théoriquement supprimé si un mauvais choix des couleurs finales le fait réappaître. Un algorithme de moins bonne qualité affichera l'image en moins de temps pour un résultat équivalent. C'est un défi de visualiser, sur un écran ne possédant que 256 couleurs affichables, une image contenant quelques millions de nuances. Le problème est de trouver (rapidement) $R = 2^q$ couleurs représentatives parmi 2^c couleurs, avec q largement inférieur à c. Mais ces 2^q couleurs représentatives ayant été trouvées, il faut ensuite déterminer, pour chaque pixel de l'écran, la couleur représentative la plus proche de la couleur de ce pixel. Pour résoudre ce problème, nous proposons une méthode basée sur les octrees et le balayage de Peano. Pour rendre plus clair notre exposé, nous définirons tout d'abord, la modélisation des couleurs utilisée [MEY86], puis le principe du balayage de Peano. Ensuite, nous présenterons les travaux antérieurs ou contemporains à notre méthode, que nous exposerons enfin.

2 Modélisation de la couleur

La perception de la couleur dépend de la nature de la lumière (considérée comme étant de l'énergie électromagnétique), de son interaction avec un matériau et de son interprétation du phénomène résultat par le système de vue humain : œil/cerveau.

L'œil humain interprète l'énergie électromagnétique dont les longueurs d'ondes sont comprises entre 400 et 800 nanomètres (10^{-9}m) comme la lumière visible. La lumière est perçue, soit directement à partir de la source lumineuse, soit par réflexion ou réfraction sur les objets. Une lumière monochromatique est une lumière dont les longueurs d'ondes sont voisines à un nanomètre près. D'un autre coté, une lumière dont toutes les composantes sont de même poids (c'est à dire toutes les longueurs d'onde visible sont d'une même force) n'a pas de couleur, elle est perçue blanche. Un objet réfléchant 80% de la lumière blanche (sans modifier la répartition des longueurs d'onde) apparait blanc, s'il réfléchit moins de 3 %, il apparait noir.

Si les longueurs d'ondes dominantes sont vers les hautes fréquences, la lumière paraîtra rouge, au milieu, verte, et dans les basses fréquences : bleue. Une couleur est définie par sa teinte, sa saturation et sa luminance. C'est une définition physiologique et non pas physique. La teinte correspond à la longueur d'onde dominante. La saturation est la

mesure de la quantité de blanc contenue dans une couleur. Une couleur pure est 100 % saturée : elle ne contient pas de blanc. Une couleur saturée à 0 % est blanche. La luminance correspond à la quantité d'énergie que reçoit l'œil.

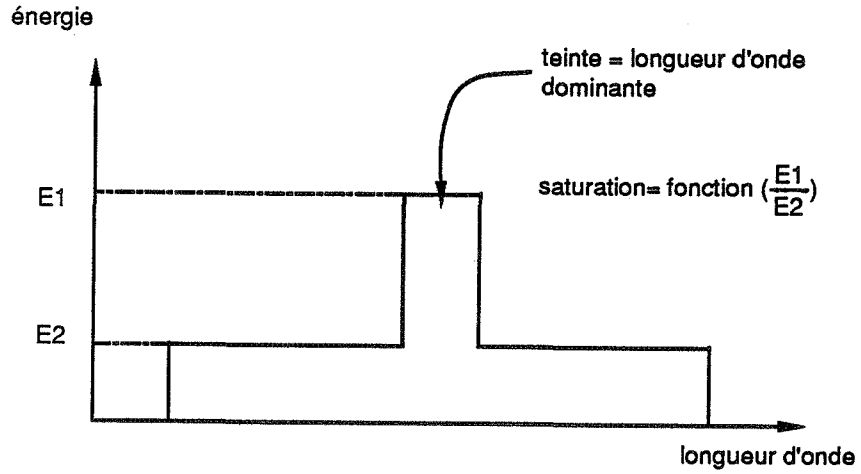


Fig. 1 saturation et teinte

Les sources lumineuses monochromatiques sont rares. Les couleurs perçues sont, en réalité, un mélange de lumière monochromatiques. La théorie de la colorimétrie est basée sur la supposition de l'existence de trois sortes de cônes dans l'œil. Chaque type de cône reconnait les longueurs d'ondes basses, moyennes et hautes comme étant du bleu, du vert et rouge. En fait, l'œil est plus sensible au vert qu'au rouge ou au bleu (ce qui serait dû à la couleur de la lumière solaire). La lumière blanche peut être produite par le mélange de trois couleurs bien choisies. C'est à dire, deux de ces couleurs ne doivent pas être capables de reproduire par mélange la troisième. On utilise, le plus souvent, les modèles RVB (Rouge Vert Bleu) ou CMJ (Cyan Magenta Jaune) [SCHW87]. Remarque : le magenta n'apparait pas dans le spectre diffracté par un prisme, c'est une pure création de l'œil. Le modèle RVB est un modèle additif, il est utilisé sur les moniteurs couleurs; en peinture, c'est le modèle soustractif CMJ qui est utilisé.

$R + V + B = \text{Blanc}$

$C + M + J = \text{Noir}$

On peut remarquer, de plus, que le cyan est le complémentaire du rouge, le magenta du vert et le jaune du bleu (deux couleurs sont complémentaires si leur mélange donne du blanc). La différence entre les systèmes additifs et soustractifs explique, en grande partie, les réticences des créatifs. En effet, habitués aux techniques picturales classiques basées sur le système soustractif, lors de leur premier contact avec des palettes graphiques, ils sont, le plus souvent, rebutés par l'approche additive. Dans le système RVB, une couleur C est obtenue par :

$C = rR + vV + bB$

L'œil humain est capable de reconnaître approximativement 350 000 couleurs. Quand les couleurs ne se différencient que par leur teinte, 128 couleurs sont discriminables. On distingue mieux dans le bleu-jaune que dans les extrêmes du spectre. Pour des différences dans la saturation, seules 16 saturations du jaune sont reconnaissables et 23 dans le rouge-violet. L'œil n'est pas capable de distinguer plus de 256 niveaux de gris. Par conséquent, une mémoire d'image qui possède 24 bits de profondeur par pixel, ce qui permet de coder des composantes rouge, verte et bleue sur 256 niveaux, est suffisante pour représenter n'importe qu'elle image. Malheureusement le prix de telles mémoires est encore, à l'heure actuelle, trop élevé pour que toute station de travail soit ainsi configurée. Pour une image 1024x1024, on a besoin de 2^{20}x3 octets, soit 3 Mega-octets. Nous travaillons, à l'Ecole des Mines de St-ETIENNE, sur une mémoire d'image qui possède 12 bits de profondeur alloués au codage de la couleur et une table de couleur qui prend en entrée 12 bits et en donne 24 en sortie. Ceci nous permet de représenter 4096 couleurs, simultanément sur un écran, parmi plus de 16 millions. Les stations de travail couleur, le plus souvent, peuvent afficher simultanément 256 couleurs parmi 16 millions, en utilisant aussi une table de couleur. Nous reprendrons, plus en détails, la définition d'une table de couleur un peu plus loin. Nous allons, tout d'abord, introduire une notion qui nous servira dans tout ce chapitre : le balayage de Peano

3 Balayage de Peano

En 1890, le mathématicien Guiseppe Peano [PEAN1890], a défini une famille de courbes qui permettent de plonger $\aleph^d$, de façon bijective, dans $\aleph$, où $\aleph$ est l'ensemble des entiers naturels et d strictement supérieur à 0. Par conséquent, on peut représenter en une seule dimension des données multidimensionnelles [LAUR85], [LAUR86]. Ces courbes, qui remplissent l'espace, ont les propriétés suivantes:

- il existe une bijection entre l'espace $\aleph^d$ et $\aleph$,
- ces courbes passent une seule fois par chacun des points de $\aleph^d$,
- elles gardent la notion de proximité : des points proches dans $\aleph^d$ sont susceptibles d'être proches sur la courbe,
- elles induisent un ordre sur les points de $\aleph^d$, dit ordre de Peano. L'indice d'un point sur cette courbe, est appelé clef de Peano.

Parmi ces courbes, deux semblent intéressantes, la première, utilisée par Stevens et al [STEV 83], [LEHA84], que l'on appellera courbe en U, garde assez bien la notion de proximité :

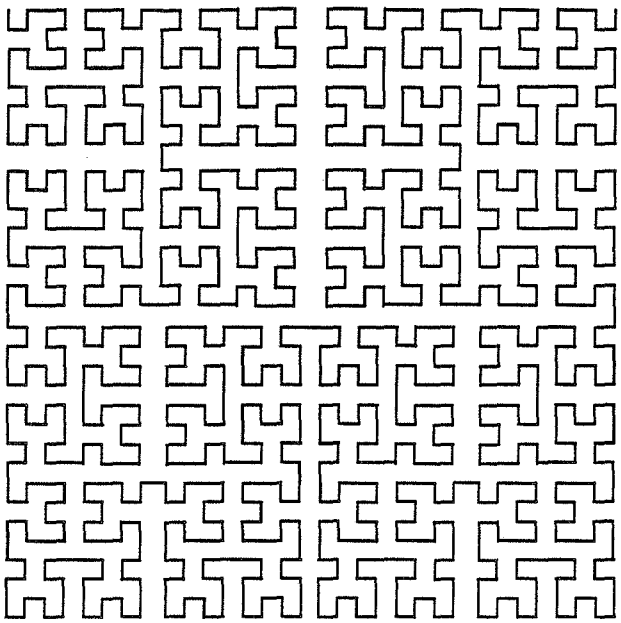


Fig 2: Courbes de Peano en U

Mais la génération de cette courbe impose des calculs compliqués et demande des temps de calcul plus élevés que la courbe suivante, que nous appellerons courbe en N. Elle possède un côté fractal très séduisant, permettant un calcul simple des clefs de Peano.

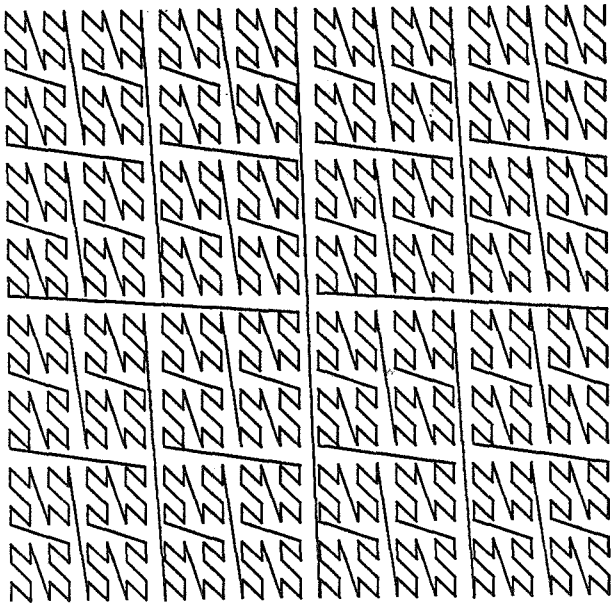


Fig 3: Courbes de Peano en N

La notion de proximité est moins bien gardée, mais les calculs de la clef de Peano

d'un élément (i,j) de $\aleph^2$ est simple. Supposons que i est pour valeur en base 2 : $i^{n-1}i^{n-2}...i^1i^0$ et j : $j^{n-1}j^{n-2}...j^1j^0$, où i^k et j^k sont des digits et n le nombre de bits utiles pour coder i et j. Leur clef de Peano peut être facilement déduit par un entrelacement de leurs bits soit :

$$p = i^{n-1}j^{n-1}i^{n-2}j^{n-2}...i^1j^1i^0j^0$$

Cet ordre étant récursif, on peut faire des regroupements locaux, il est donc bien adapté à la manipulation de quadtree et d'octree.

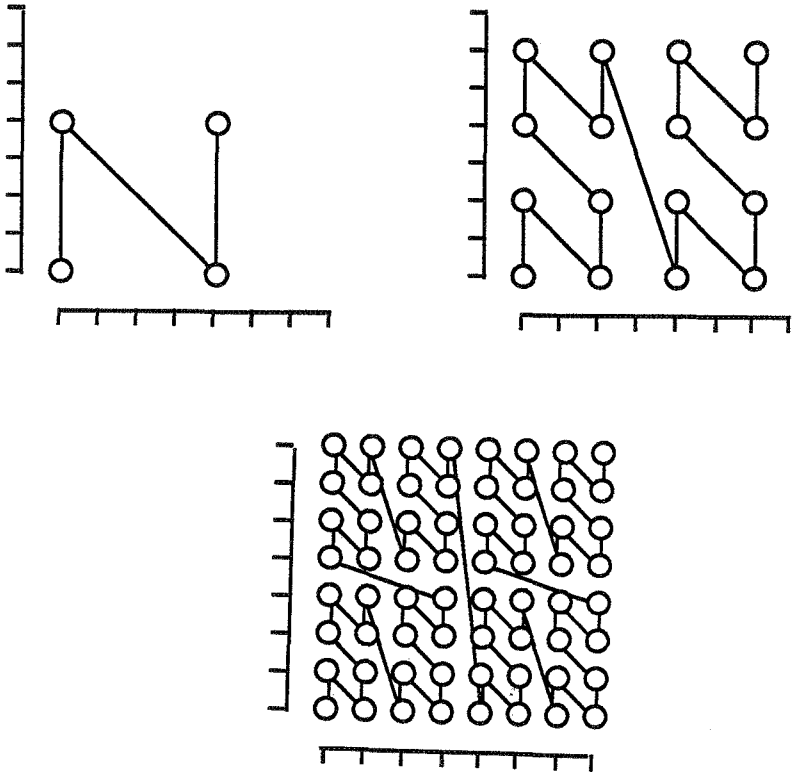


Fig 4: courbe en N représentant l'ordre des clefs,
selon la finesse du niveau de résolution

Chaque niveau d'un quadtree peut être linéarisé par une courbe de Peano, ce qui permet la définition suivante : soit p une clef de Peano associée à une cellule d'un quadtree, on peut retrouver la clef de Peano de ces ancêtres en supprimant les bits de poids faible de p. Prenons un exemple, i = 7 = (0111), j = 3 = (0011), ce qui entraîne que p = 47 = (00101111). La clef du père de p est égale à (p/4)*4 = 44, son grand-père est égal à (p/16)*16 = 32, son arrière-grand-père à (p/32)*32 = 0 et son ultime ancêtre à (p/64)*64 = 0. La division que nous utilisons est bien entendu la division entière.

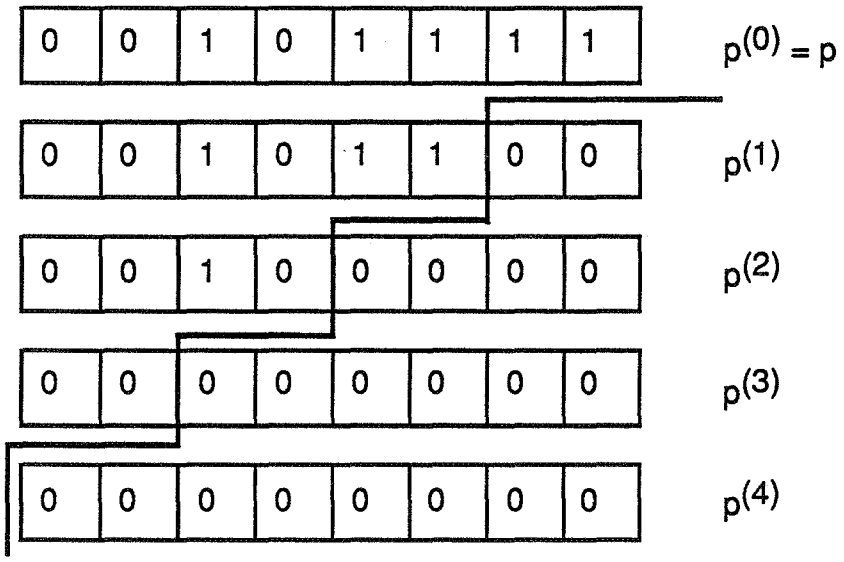


Fig 5: les ancêtres de p = 47

Nous obtenons ces ancêtres par un masquage des bits de poids faible (très facilement mis en œuvre avec le langage C).

$p \& 0xFF = p^{(0)} = p = 47$

$p \& 0xFC = p^{(1)} = 44$

$p \& 0xF0 = p^{(2)} = 32$

$p \& 0xC0 = p^{(3)} = 0$

$p \& 0x00 = p^{(4)} = 0$, où & est l'opérateur "et" sur les bits,

et 0xHH la notation hexadécimale d'un nombre.

Dans le cas d'un octree, nous définissons de la même manière la clef de Peano déduite d'un triplet (i,j,k) :

$p = i^{n-1}j^{n-1}k^{n-1}i^{n-2}j^{n-2}k^{n-2}...i^{1}j^{1}k^{1}i^{0}j^{0}k^{0}$.

Les ancêtres se retrouvent de façon aussi simple par masquage, nous utiliserons une notation octale des constantes de masquage :

$p \& 07777 = p^{(0)} = p$

$p \& 07770 = p^{(1)}$

$p \& 07700 = p^{(2)}$

$p \& 07000 = p^{(3)}$

$p \& 00000 = p^{(4)}$

4 Table de couleurs

Il est très difficile de déterminer pour une image qui peut comporter jusqu'à 512x512 couleurs soit 2^{18} pour une image 512x512 pixels, quelles doivent être les 4096 à 64 couleurs représentatives qui seront contenues dans une table de couleurs. Le problème est de trouver (rapidement) $R = 2^q$ couleurs représentatives parmi 2^c couleurs, avec $q << c$. Mais ces 2^q couleurs représentatives ayant été trouvées, il faut ensuite déterminer, pour chaque pixel de l'écran, la couleur représentative la plus proche de la couleur de ce pixel. Nous allons maintenant donner la définition d'une table de couleurs. A chaque point de l'écran, on associe, non plus une couleur RVB, mais une adresse dans la table des couleurs. Cette adresse, ou indice, dans la table permet de retrouver la couleur correspondante.

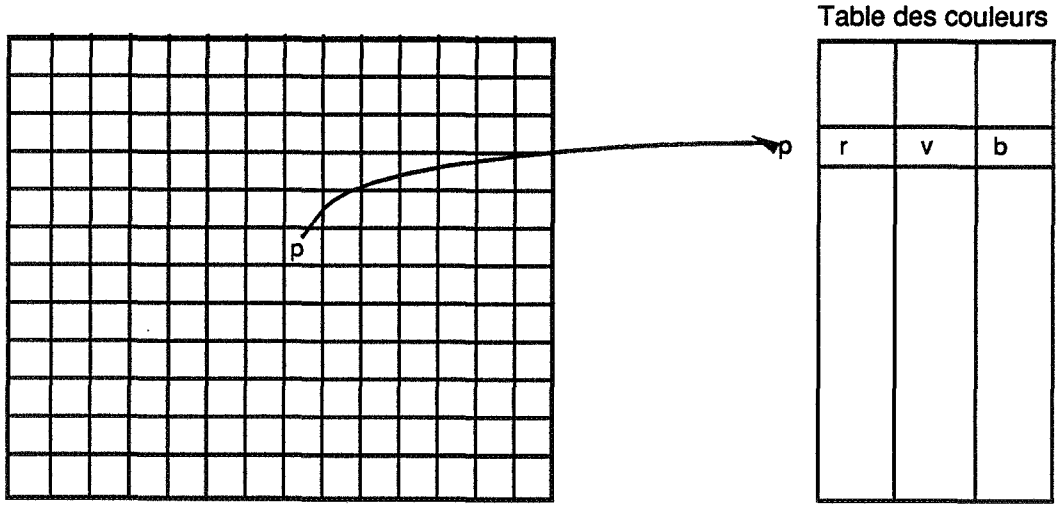


Fig. 6 indexation d'une image à travers une table de couleurs

Une table de couleurs permet aussi d'autres effets :

- si, au lieu d'assigner une couleur à travers la table de couleurs, on donne le numéro caractéristique d'un objet, cela permet, très simplement, en désignant un pixel, de retrouver l'objet qui se projette sur ce pixel. On a, par conséquent, une bijection entre les adresses de la table et les objets de la scène.

- on peut modifier, très facilement et très rapidement, les couleurs d'une image en changeant simplement la table de couleurs. Ce qui permet des effets psychédéliques souvent assez réussis. Si la scène ne possède que peu d'objets, on peut simuler de façon grossière une animation [SHOU 79].

- en modifiant la table de couleurs, on peut simuler le déplacement d'une source lumineuse [WARN 83], avec la contrainte de n'utiliser que peu de couleurs.

Nous allons maintenant présenter une solution simple au problème du choix des R couleurs représentatives.

4.1 Une solution simple

Une solution simple, mais qui donnera des résultats grossiers, consiste à remplir la table en choisissant équitablement des représentants dans tout l'espace des couleurs. Cette méthode simple peut être faite de la façon suivante : soit q le nombre d'adresses dans la table, on peut supposer, pour simplifier, que q = 3s.

Les couleurs représentatives contenues dans la table seront :

$r=(\varepsilon_{r,c-1}\varepsilon_{r,c-2}\varepsilon_{r,c-3}...\varepsilon_{r,c-s}0...0)$,

$v=(\varepsilon_{v,c-1}\varepsilon_{v,c-2}\varepsilon_{v,c-3}...\varepsilon_{v,c-s}0...0)$,

$b=(\varepsilon_{b,c-1}\varepsilon_{b,c-2}\varepsilon_{b,c-3}...\varepsilon_{b,c-s}0...0)$,

avec $\varepsilon_{r,i}= 0,1$; $\varepsilon_{v,i} = 0,1$; $\varepsilon_{b,i}= 0,1$ pour $0 \leq i<c$

Cette couleur sera mise dans la table à l'adresse ad suivante :

$ad = \varepsilon_{r,c-1}\varepsilon_{r,c-2}\varepsilon_{r,c-3}...\varepsilon_{r,c-s}\varepsilon_{v,c-1}\varepsilon_{v,c-2}\varepsilon_{v,c-3}...\varepsilon_{v,c-s}\varepsilon_{b,c-1}\varepsilon_{b,c-2}\varepsilon_{b,c-3}...\varepsilon_{b,c-s}$

On a bien $2^s x 2^s x 2^s = 2^q$ couleurs représentatives équitablement réparties. Cela signifie que l'information principale se trouve dans les bits de poids forts. Prenons maintenant, une couleur contenue dans une image, soit :

$r=(r_{c-1}r_{c-2}r_{c-3}...r_0)$, $v=(v_{c-1}v_{c-2}v_{c-3}...v_0)$, $b=(b_{c-1}b_{c-2}b_{c-3}...b_0)$,

A cette couleur sera associée l'adresse obtenue en gardant uniquement les bits de poids fort, c'est à dire :

$ad=r_{c-1}r_{c-2}r_{c-3}...r_{c-s}v_{c-1}v_{c-2}v_{c-3}...v_{c-s}b_{c-1}b_{c-2}b_{c-3}...b_{c-s}$

Cette méthode est très rapide mais donne de mauvais résultats. En effet, tous les représentants n'existeront peut être pas sur l'image. Des entrées dans la table ne seront peut être pas utilisées alors que des zones qui présentent des dégradés seront très mal représentées. Pour avoir une meilleure visualisation, il faut que les couleurs contenues dans la table dépendent de l'image à visualiser. Si cette image est dans une gamme de rouge et vert et qu'aucun bleu n'intervient, il ne faut aucun bleu dans la table.

Le problème du choix de ces 2^q couleurs apparait. Nous allons tout d'abord présenter les différentes méthodes existantes. Puis nous introduirons des notions nécessaires pour notre méthode que nous exposerons, ensuite, en détail .

4.2 Rappels des méthodes existantes

Heckbert présente deux types de quantification adaptative d'images [HECK82]. La première, basée sur les travaux de Boyle et Lippman [HECK80], s'appelle algorithme de popularité (popularity algorithm), la seconde, algorithme de partage équitable (median cut algorithm). Ces deux méthodes se décomposent en quatre phases :

1 construction d'un histogramme des fréquences des couleurs dans l'image, en tronquant les couleurs initialement codées sur 3x8 bits à 3x5 bits pour que la taille du tableau contenant l'histogramme soit limitée à 32768.

2 choix de la table de couleurs selon la distribution des couleurs dans l'image. Les 2^q couleurs les plus fréquentes pour l'algorithme de popularité. Pour l'algorithme de partage équitable, le cube des couleurs est divisé en 2^q boîtes rectangulaires contenant le même nombre de couleurs sans que leur fréquence d'apparition sur l'écran ne soit prise en compte, un représentant de chacune des ces boîtes est ensuite choisi.

3 calcul d'une fonction qui plaque les couleurs 24-bits sur les couleurs représentatives.

4 re-calcul de l'image en changeant la couleur de chaque pixel en une entrée dans la table.

La première méthode donne de bien plus mauvais résultats que la seconde (sauf sur certaines images bien particulières : pas trop de couleurs ni trop peu). Le première méthode engendre des erreurs pour des couleurs peu nombreuses en pixels mais qui peuvent représenter des détails importants de l'image. Les zones présentant des larges dégradés gagnent aux dépens des détails. La deuxième donne un meilleur rendu, mais l'exactitude des couleurs, par rapport à l'image en 24 bits, n'est pas très bonne. En effet, supposons qu'il y ait P pixels presque tous bleus sauf quelques uns rouges ou verts par exemple : $\frac{P}{2^{q-1}}$ pixels rouges (250,0,0) et $\frac{P}{2^{q-1}}$ pixels verts (0,253,0), les $(N-\frac{N}{2^{q-2}})$ pixels restants seront supposés dans les bleus. Une seule boîte regroupera les pixels rouges et verts. Une moyenne des couleurs rouges et vertes donnera du jaune, ainsi ni les pixels rouges ni les pixels verts ne seront bien représentés. Cet exemple est bien évidemment exagéré, mais il est symptomatique du comportement général de l'algorithme, aucune barrière n'empêche des mélanges abusifs de couleurs. De plus, la place mémoire nécessaire et les temps de calculs sont bien plus grand que pour les méthodes que nous allons exposer maintenant.

La méthode que nous présentons maintenant est due à Stevens et al.[STEV 83]. Elle est basée sur le balayage de Peano. L'ordre de Peano associe de façon bijective, et donc réversible, à tout $(r,v,b) \in \aleph^3$ une clef de Peano $p \in \aleph$. Cette clef garde à peu près la notion de proximité. Ce qui permet, à partir des clefs de Peano de l'image codée sur 15 bits de construire un histogramme des fréquences des clefs. Ensuite ils procèdent en deux

étapes :

1 toutes les clefs dont la fréquence dépasse le nombre de pixels divisés par le nombre de couleurs de la table de couleurs sont choisies.

2 l'histogramme des clefs de Peano est divisé en (2^q - nombre de couleurs déjà choisies) parties contenant le même nombre de pixels. Dans chacune des ces parties le milieu est pris comme représentant.

Pour chaque clef choisie on peut retrouver la couleur qui lui est associée. la table des couleurs est donc effectivement calculée. Maintenant, pour chaque pixel on calcule sa clef de Peano est on retrouve la clef représentante la plus proche.

Comme les algorithmes de popularité et de partage équitable, cette méthode est efficace lorsque qu'il y a peu de couleurs, ou peu d'entrées dans la table des couleurs (<256). La fonction de balayage qu'ils ont choisie est longue et difficile à calculer. De plus, prendre le représentant égal au milieu, peut amener de grossières erreurs, car à nouveau aucune barrière n'empêche des mélanges abusifs.

Coquillart [COQU 84], utilise la binarisation d'un octree sans le dire explicitement. En effet, chacun des huit fils d'un nœud de niveau i est associé de façon unique aux $i^{ème}$ bit des composantes rouge, verte et bleue (en utilisant ce qui revient à une clef de Peano). On a donc une division récursive en huit sous-cubes du cube des couleurs. Le principe de l'algorithme est le suivant :

POUR chaque point de l'image FAIRE
SI la couleur du point n'existe pas dans la structure
l'insérer.
FSI
TANT QUE le nombre de feuilles est supérieur à 2^q
supprimer une feuille.
FTANTQUE
FPOUR

Les destructions sont exécutées de façon ordonnée. Pour faciliter la destruction des feuilles, il est nécessaire de mémoriser un pointeur par niveau, pour rendre possible le balayage de l'arbre en préordre. La destruction d'une feuille revient, en fait, à représenter la couleur avec un bit de moins pour chaque composante, c'est à dire tronquer la valeur. Toute nouvelle couleur est insérée avec une précision compatible avec la profondeur courante de l'arbre. La liste des couleurs qui doivent être présentes dans la table est directement déductible du chemin que l'on parcourt de la racine à chacune des feuilles. Pour construire l'image finale, l'indice, dans la table, de chacune des couleurs représentées est stocké dans chaque feuille de l'arbre. Donc pour chacun des pixels, ils suffit de

parcourir l'arbre pour trouver l'indice dans la table des couleurs.

Cette méthode ne prend pas beaucoup de place, même si chaque feuille garde un indice. Mais cette méthode ne tient pas compte de la fréquence d'apparition d'une couleur. Les bits de poids faible sont supprimés, ce qui entraîne une perte d'information. Prenons une couleur :

r = 0001 1111 v = 0001 1111 b = 0001 1111

Supposons que l'information soit conservée sur 3 bits pour chacune des composantes, et que les couleurs les plus proches soient :

r1 = 0000 0000 v1= 0000 0000 b1 = 0000 0000

r2 = 0010 0000 v2 = 0010 0000 b2 = 0010 0000

Par cette méthode, la couleur la plus proche de (r,v,b) sera (r1,v1,b1) dans l'arbre , alors que dans l'espace des couleurs c'est (r2,v2,b2) qui est bien plus proche. En effet, toute couleur contenue dans le sous-cube $0 \leq r < 32, 0 \leq v < 32, 0 \leq b < 32$ se projette sur (r1,v1,b1). C'est le problème entraîné par l'utilisation d'octree. Mais dans cette méthode, la décision de tronquer les couleurs au lieu de garder le centre du sous-cube ou mieux la moyenne des couleurs dans ce sous-cube, aggrave cet artefact.

La technique que nous allons présenter maintenant est basée sur l'utilisation d'un octree pour le stockage des couleurs [GERV88]. Comme tous les algorithmes précédents, le déroulement se fait en trois phases :

1 construction d'une structure de données, qui permettra de trouver les représentants,

2 remplissage de la table des couleurs,

3 mise en relation des couleurs de l'image avec les couleurs représentatives.

Le principe de cette méthode est le suivant : l'image est lue séquentiellement. Les R premières couleurs sont stockées dans un octree. Si une nouvelle couleur, n'existant pas dans l'octree, doit être ajoutée, des couleurs proches sont regroupées et substituées à leur moyenne. Pour toute nouvelle couleur, cette dernière étape est répétée. On est donc assuré, à la fin de la construction de l'octree qu'il n'existera pas plus de R couleurs simultanément dans l'octree. Par conséquent, la taille mémoire nécessaire est limitée. Deux couleurs seront considérées proches, si elle font parties du même nœud de l'arbre, c'est à dire du même sous-cube. La réduction de l'octree lors de l'arrivée d'une $(R+1)^{ième}$ couleur, se fera selon les critères suivants :

- parmi tous les nœuds pouvant être réduits, ceux qui sont à la plus grande profondeur dans l'octree seront les premiers choisis, puisque les couleurs qu'ils représentent sont les plus proches (dans l'octree).

- s'il existe plus d'un nœud de profondeur maximale, d'autres critères pourront être

utilisés pour une sélection optimale :

- réduire les nœuds qui représentent le moins de pixels. De cette façon, la somme des erreurs de représentation pour chaque pixel sera maintenue faible.

- réduire les nœuds qui représentent le plus de pixels. Ainsi, les larges zones seront uniformément remplies avec une couleur légèrement fausse, mais les détails (comme l'antialiassage) seront préservés.

Une partie appréciable des temps d'exécution est passée à la recherche d'un nœud optimal pour chaque réduction. Pour optimiser les temps de calcul, les auteurs proposent de conserver 8 listes linéaires des réductibles, correspondant aux 8 niveaux de l'octree. Tous les nœuds de même profondeur sont dans une même liste. Les nœuds de profondeur maximale peuvent donc être trouvés facilement, mais cela demande d'augmenter la taille de l'octree en ajoutant un pointeur supplémentaire à chaque nœud. Ces listes induiront vraiment un gain de temps, si un nouveau nœud est inséré en début de liste, et sur le nœud détruit est le premier de la liste. Ce qui revient à dire que ces listes sont des piles. Par conséquent, les critères de choix optimal de réduction présentés plus haut, ne peuvent plus être utilisés. En effet, pour chercher le nœud comportant le plus de pixels, par exemple, il est plus rapide de comparer les valeurs en nombre de pixels de tous ses pères dans l'octree, dans le pire des cas 64 tests (l'octree a une profondeur maximale de 8 et 8 fils à chaque noeud), que de tester, dans le pire des cas, les R nœuds d'une liste (ex. R = 64 à 4096).

Chaque nœud contient un champ supplémentaire pour indiquer l'indice qui lui correspondra dans la table des couleurs. La recherche de chaque couleur de l'image dans l'octree s'arrêtera dans une feuille. Ce nœud contiendra une couleur proche de la couleur initiale, et donc une couleur représentative.

4.3 Quantification des couleurs par codage dans un octree avec le balayage de Peano

La méthode que nous proposons se décompose en trois phases :

1 Pour chacun des pixels, ajout de sa clef de Peano dans l'octree

2 Calcul de sa table des couleurs représentatives

3 Recherche, pour chaque pixel de l'écran, de la couleur la plus proche.

En réalité, nous avons étudié deux variantes. La première travaille sur peu de couleurs représentatives (≤ 256), en essayant d'avoir les couleurs les plus représentatives aux dépens de la place mémoire nécessaire. La seconde, par contre, essaie de minimiser la quantité de mémoire nécessaire, sans trop de perte de temps, dans le cas où le nombre de couleurs représentatives est grand, 4096 par exemple.

4.3.1 Première approche

Nous ne pouvons afficher, au plus, que R couleurs, R ≤ 256. Par conséquent, les couleurs représentatives doivent être les plus exactes possibles. Nous conserverons, donc dans chaque feuille la couleur moyenne qui lui est associée, et le nombre de pixels qu'elle représente. Pour chaque niveau dans l'arbre, nous gardons, le nombre de nœuds non vides, ce qui revient à dire, le nombre de chemin de la racine de l'octree jusqu'à ce niveau. Pour une notation plus cohérente, la profondeur 7 caractérisera la racine de l'octree, tandis que la profondeur 0 sera associée aux feuilles.

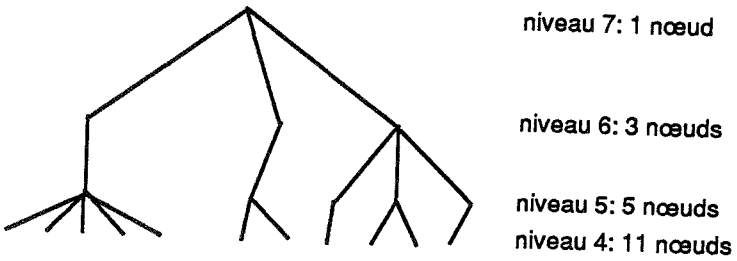


Fig.7 : nombre de nœuds par niveau

Dans une première étape, tant qu'il n'y a pas plus de R couleurs au niveau 0, on ne construit pas l'octree. Ce qui évite de construire une structure complexe pour une image qui finalement n'en aura pas besoin. Ces couleurs sont stockées dans une table par une fonction de dispersion pour un accès rapide. Cette table contient aussi le nombre de pixels associés à chacune de ces couleurs. Dès que le nombre de couleurs dépasse R, on initialise l'octree en créant les R feuilles qui contiennent, alors, ces couleurs et le nombre de pixels qui leur sont associés.

Nous appellerons niv, le niveau le plus profond dans l'octree. Au départ niv vaut 0. On rajoute des couleurs dans l'octree tant que le nombre de noeuds au niveau (niv+1) est inférieur à R. Dès que ce seuil est atteint, on supprime toutes les feuilles du niveau niv, en calculant la moyenne des couleurs "frères", en affectant à leur père cette moyenne puis en libérant la place occupée par ces feuilles. On affecte la valeur niv + 1 à niv.

Pour décrire l'octree, nous définissons la structure suivante, en C.

```
struct T_OCTREE {
        int r,v,b;
        int nb_pixels;
        struct T_OCTREE fils[8];
        }
```

L'algorithme de remplissage est donc le suivant :

```
   creer_coul(p,r,v,b,octree)
{
/ *p est la clef de Peano de la couleur  (r,v,b)
  ************************************/
SI nb_nœud(niv+1) > R
ALORS
        supprimer(niv);
        niv = niv + 1;
FSI
profondeur = 7;
ajouter_coul(niv,p,r,v,b,octree,profondeur);
}


ajouter_coul(niv,p,r,v,b,octree,profondeur)
{
SI (profondeur = niv)
ALORS
        SI (octree est vide)
        ALORS
                allouer(octree);
                nb_nœud[niv] = nb_nœud[niv] + 1;
        FSI
        mettre(r,v,b,octree->fils[(p>>profondeur)&07]);
SINON
        SI (octree est vide)
        ALORS
                allouer(octree);
                nb_nœud[profondeur] = nb_nœud[profondeur] + 1;
        FSI
        ajouter_coul(niv,p,r,v,b,octree->fils[(p>>profondeur)&07]
                     ,profondeur-1);
FSI
}


metttre(r,v,b,octree)
{
octree->r  = octree->r + r;
octree->v = octree->v + v;
octree->b = octree->b + b;
octree->nb_pixels = octree->nb_pixels + 1;
}
```

Choix des couleurs représentatives

Il faut minimiser les erreurs de couleurs pour qu'il n'y ait pas de défauts choquants : une orange qui devient bleue (quoique ...). Soit *C* le nombre nœuds au niveau *niv + 1* et *A* le nombre de nœuds au niveau *niv*.

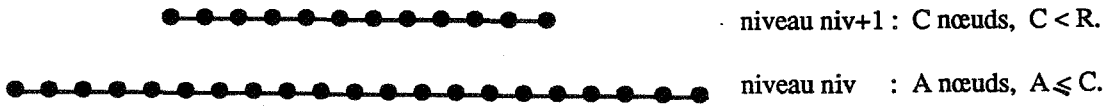


Fig. 7 les noeuds des niveaux niv et niv+1

Pour être sûr, que toute couleur de l'image aura un représentant qui ne sera pas plus éloigné que la diagonale d'un cube de taille 2^{niv+1}, on prendra obligatoirement les P couleurs associées aux P nœuds du niveau *niv* + 1. Mais, nous avons le choix pour les (R -P) couleurs qui restent. Nous pouvons, donc, les choisir parmi les A couleurs des A nœuds du niveau niv. Comme nous voulons minimiser l'erreur, et que nous estimons que, par rapport à mauvais dégradé, l'aliassage est un moindre mal : un observateur est remarquera d'abord un mauvais dégradé avant de critiquer l'aliassage. Nous choisirons, par conséquent, les (R-P) couleurs les plus fréquentes parmi A. Une autre justification de ce choix est que les couleurs qui représentent peu de pixels sont certainement bien plus nombreuses que celles qui en représentent beaucoup. Par conséquent, on peut se consoler en se disant que l'antialiassage serait quand même mal rendu. Nous privilégirons les hautes intensités, car l'œil est plus sensible à ces dernières qu'aux basses intensités. Ce qui se met facilement en œuvre par le choix de l'ordre de parcours de l'octree.

En réalité l'algorithme que nous utilisons est le suivant :

```
{
/* A est le nombre de couleurs Ancêtres (du niveau niv+1) */
/* R est le nombre de couleurs Représentables */
/* P est le nombre de couleurs Présentes dans l'octree (du niveau niv)
*/

SI P ≤ R {
        prendre_toutes_les_couleurs(niv);
        }
SINON   {
        choix = R - A;
        TANT QUE choix ≠ 0 {
           prendre_la_couleur_la_plus_présente(niv);
           A  =  nombre_de_couleurs_du_niveau(niv+1);
           R = R - 1;
           choix = A - R;
           }
        SI A ≠ 0     {
           prendre_toutes_les_couleurs(niv+1);
           }
        }
}
```

Travailler sur deux niveaux entraîne une quantité de mémoire majorée par *Rx8xtaille_d'une_cellule*, soit dans le pire des cas, 96ko pour R = 256, ce qui n'est pas prohibitif. Par contre les temps de calculs sont améliorés, par rapport à la méthode de Gervautz et al., puisque la suppression des feuilles se fait une fois par niveau, sans critères de suppression, et non pas pour chaque dépassement de R. Nous remarquons aussi, qu'au lieu de chercher les couleurs à supprimer nous cherchons les couleurs à garder. Ces deux recherches n'ont pas le même ordre de complexité, la première est de l'ordre de $\text{niv\_sup}\frac{R(R-1)}{2}$ (où niv_sup est le nombre de niveaux supprimés), tandis que l'autre est de l'ordre de 8niv_supR.

Mais ces méthodes, quand le nombre de couleurs représentables est grand, peuvent demander trop de place pour être utilisable. Nous avons développé une version qui travaille sur plus de 256 couleurs (en général 4096).

4.3.2 Méthode pour plus de 256 couleurs

Dans ce cas nous voulons limiter la place mémoire nécessaire. Nous utiliserons à peu près la méthode précédente, mais l'octree sera réduit. On ne conservera que le nombre de pixels, la couleur ne sera plus gardée. En effet, elle intervient de façon implicite dans la

stucture de l'octree. Ne pas garder explicitement la moyenne des couleurs permet de faire un important gain de place : soit F le nombre de noeuds et de feuilles, dans la méthode précédente F*48 octets sont nécessaires. Si on supprime l'information de couleur on a besoin de F*36 octets, soit un gain d'un quart. Par contre, la fidélité sera moins bonne puisque l'on perd de l'information. Mais nous verrons sur les images en annexe que la différence est minime. La structure de l'octree est la suivante :

```
struct T_OCTREE {
        int nb_pixels;
        struct T_OCTREE fils[8];
        }
```

L'octree est construit de la même manière que précédemment. Le choix des R feuilles répresentatives est identique. Mais on remédie à la perte d'information sur la couleur en choisissant le mieux le représentant d'une feuille. Une feuille de l'octree représente un cube dans l'espace des couleurs. On choisi le centre de ce cube pour représentant. Mais nous travaillons dans un espace discret, le centre d'un cube est en réalité les huit couleurs les plus proches du centre réel (ou seulement une si le cube est réduit à une couleur). Le choix d'une de ces huit couleurs est arbitraire. Nous choisissons la plus grande de ces couleurs (au sens de la norme infinie). Cette couleur peut être facilement calculée en utilisant le balayage de Peano :

> soit p la clef de Peano d'une feuille et i son niveau, la clef de peano du centre sera le septième fils de p ie (p | (07 << i))

5 Conclusion

Nous avons proposé deux nouvelles méthodes donnant de bons résultats qui ne prennent pas trop de place, en des temps négligeables par rapport aux performances du tracé de rayons. Les images visualisées avec seulement 256 couleurs perdent beaucoup de leur réalisme. Il convient alors d'utiliser conjointement une méthode de tramage permettant de simuler beaucoup plus de couleurs, [PURG88], en jouant sur les hautes définitions offertes, de nos jours, par les stations de travail.

Chapitre 5

SKY : un logiciel de tracé de rayons

Dans ce chapitre est décrit de façon plus précise le logiciel de tracé de rayons SKY intégrant les techniques exposées dans les chapitres 2, 3 et 4. Ce chapitre se décompose en trois parties. La première présente le langage de saisie de données utilisé : Castor. Une extension écrite en C++ de ce langage est ensuite exposée. La deuxième partie traite les options et les techniques de visualisation utilisées dans SKY. La troisième et dernière partie aborde le problème de la parallélisation du tracé de rayons en présentant une réalisation de la parallélisation de SKY sur une machine hôte parallèle : Capitan©.

1 Saisie des données

1.1 Le langage de saisie Castor

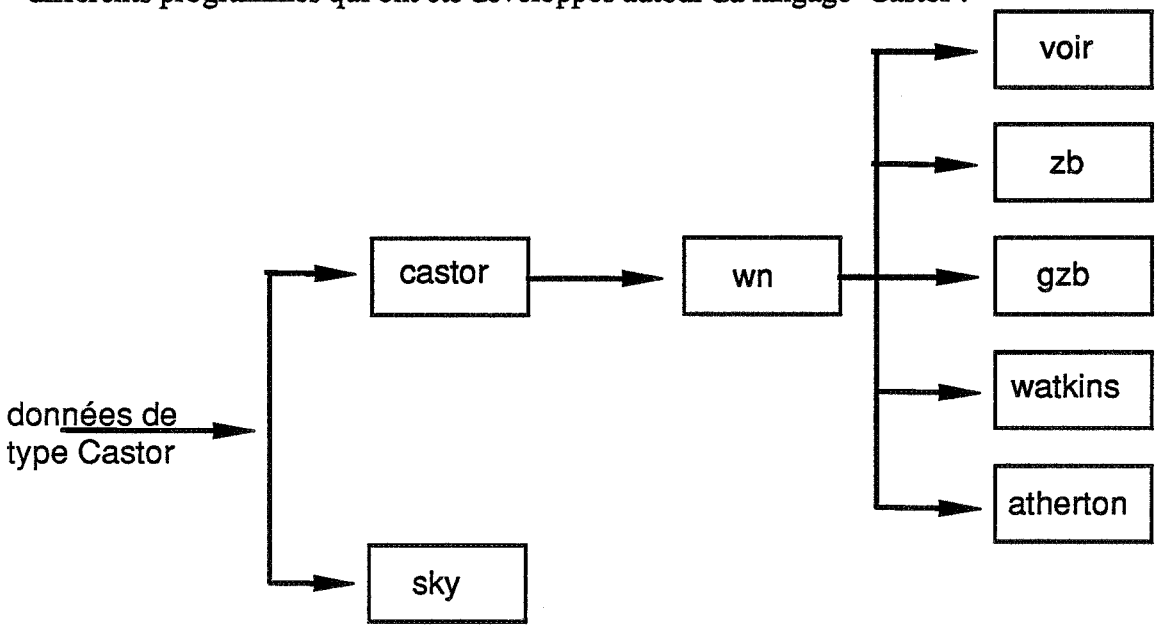
1.1.1 Choix du modèle CSG

En CAO, le modèle le plus utilisé à l'heure actuelle est le modèle B-Rep ou représentation par frontière. La frontière "exacte" des solides est explicitement connue. Ces logiciels travaillent sur des représentations polyédriques des objets de la scène. Cette modélisation permet une visualisation rapide aux dépens du réalisme (une sphère devient polyédrique). Or, pour un algorithme de tracé de rayons, il est plus rapide de calculer l'intersection d'une droite et d'une sphère en utilisant son équation, que de calculer les intersections avec les centaines de facettes composant l'approximation polyédrique. Le tracé de rayons n'a pas besoin de connaître explicitement la position d'une surface, la connaissance implicite (par une formule) est suffisante. Le degré de complexité d'un objet ne se mesure plus, pour le tracé de rayons, par un nombre de facettes, mais par le nombre ou le degré d'équations à résoudre. La modélisation par arbre de construction convient parfaitement à l'algorithme de tracé de rayons. Dans l'exemple de modélisation de terrains, la représentation de surplombs (tunnels, mines, excavations ...) est un problème complexe dans le cadre d'une représentation par B-Rep ou par grille altimétrique. En effet, la complexité (en nombre de facettes) d'un terrain est telle que les algorithmes de visualisation simplifient la détermination des faces cachées en supposant l'inexistence de surplomb, ce qui permet de déterminer un certain ordre sur les facettes ([WILL72], [WRIG73], [ANDE82]). Les opérations booléennes permettent à partir d'un terrain sans surplomb de creuser très simplement des excavations. On construit un tunnel plein puis on le retranche au terrain. Cette opération ne sera effectuée que localement pour certains pixels. Un arbre CSG a en plus l'avantage de la concision dans le cas de surfaces courbes.

© Matra

1.1.2 Les logiciels de visualisation disponibles à l'EMSE

La figure suivante, tirée de [BEIG88], présente le schéma d'organisation des différents programmes qui ont été développés autour du langage Castor :



Chacune des boîtes représente un programme :

castor est l'interprète du langage de modélisation Castor, son entrée est un fichier écrit selon la syntaxe Castor et sa sortie est un fichier de polygones; il a été écrit par Michel Beigbeder;

wn est le programme effectuant la transformation perspective et le fenêtrage des polygones; il a été écrit par Michel Beigbeder;

voir implante l'algorithme d'élimination des parties cachées de Michelucci [MICH87]; il a été écrit par Dominique Michelucci;

zb envoie les polygones vers la mémoire d'image Lexidata Solidview dans laquelle l'élimination des parties cachées est faite grâce à la méthode du z-buffer; il a été écrit par Michel Beigbeder;

gzb implante l'algorithme d'élimination des parties cachées avec antialiassage du "gz-buffer"; il a été écrit par Michel Beigbeder et Djamchid Ghazanfarpour [BEIG87];

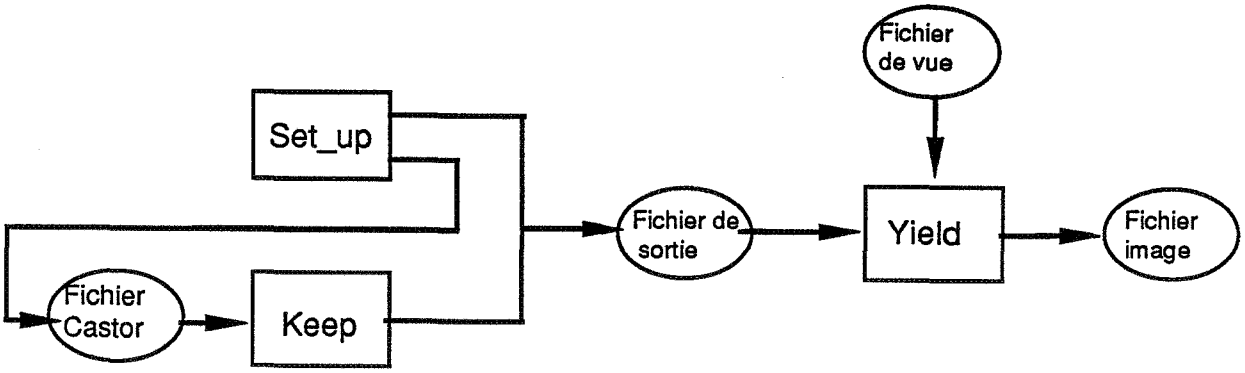
watkins implante l'algorithme d'élimination des parties cachées; il a été écrit par Jean-Michel Moreau;

atherton implante l'algorithme d'élimination des parties cachées pour la modélisation CSG d'Atherton [ATHE83] ; il a été écrit par Dominique Michelucci et Didier Tallot;

SKY est un programme de synthèse complet utilisant la méthode du tracé de rayons; il a été écrit par Jacqueline Argence;

castor et *SKY* utilisent tous les deux en entrée un fichier de description Castor. Tous les autres échanges symbolisés sur cette figure concernent des polygones.

Le programme SKY se décompose de la manière suivante :



SKY

Les trois parties qui composent ce logiciel ont les fonctions suivantes :

Keep: analyse le fichier de données écrit en Castor et produit en sortie un fichier qui contient la description de la base de données dans la structure de données interne Yield

Set_up: extension en C++ du langage Castor, il peut produire en sortie soit un fichier de type Castor soit un fichier de type Yield

Yield: programme de visualisation en tracé de rayons, il prend en entrée une base de données de type Yield, et des options de visualisation contenues dans un fichier de vue. En sortie il produit un fichier contenant la description d'une image

Nous allons maintenant décrire Castor le langage de base utilisé pour représenter une scène sous forme d'arbre CSG, puis nous présenterons l'extension C++ de ce langage.

1.1.3 Keep : Analyseur du langage de base Castor

Nous allons tout d'abord décrire les caractéristiques de base du langage Castor défini par Beigbeder [BEIG88], puis nous présenterons les améliorations apportées. Les primitives de base suivantes sont disponibles :

le cube (cu), le cône (co), la sphère (sp), le cylindre (cy), le prisme (pr).

A ces primitives sont associées une couleur (r,v,b), $0 \leq r,v,b \leq 255$, et un coefficient de spécularité[1] t. La syntaxe est par exemple la suivante :

!sp(r,v,b,t)

Le prisme a une syntaxe légèrement différente :

!pr(x_1,y_1,x_2,y_2,...,x_n,y_n,r,v,b,t) où (x_1,y_1,0), (x_1,y_1,1), (x_2,y_2,0), (x_2,y_2,1), ..., (x_n,y_n,0) et (x_n,y_n,1) sont les coordonnées des sommets du prisme.

Sur ces primitives, on peut effectuer des transformations affines :

rotation (r), affinité (a), homothétie (h), symétrie (s) , translation (t)

La syntaxe d'une rotation est par exemple :

@r(x,y,z,$\propto$) où (x,y,z) définit la droite de rotation, $\propto$ est l'angle de rotation.

Des objets sont construits à partir des primitives en utilisant des opérations ensemblistes, union (U), intersection (I), différence (D). Par exemple, une table composée d'un plateau et de quatre pieds s'écrit de la manière suivante :

```
plateau = @a(7,5,0.5) !cu(180,90,80,0);
pied    = @a(0.01,0,01,3.5) @t(0,0,-1) !cu(180,90,80,0);
table   = $U(
            plateau,
            pied,
            @t(7,0,0) pied,
            @t(7,5,0) pied,
            @t(0,5,0) pied,
            );
> table;      /* demande de visualisation */
```

Le préprocesseur du langage C permet de définir des constantes, de faire des méta-remplacements, d'inclure des fichiers ...

Pour les besoins du tracé de rayons et pour un plus grand réalisme, un certain nombre de primitives ont été rajoutées. Elles ont été définies en utilisant une syntaxe

[1] extension pour le tracé de rayons, en Castor ce paramètre, normalement dédié aux textures, n'est pas utilisé. Nous avons besoin de plus de données pour définir une texture, nous verrons cette définition un peu plus loin

semblable à celle des autres primitives:

l'hyperboloïde (hp), le cercle (ci), le carré (sq), le polygone (po), le feuillage (tr), le terrain (ld).

L'hyperboloïde, le carré, le cercle et le feuillage s'utilisent de la même manière qu'une sphère, le polygone de la même manière qu'un prisme, par contre le terrain a une syntaxe différente :

!ld(nom_terrain,r_b,v_b,b_b,t_b,r_m,v_m,b_m,t_m,r_h,v_h,b_h,t_h) où nom_terrain est le nom du fichier contenant la description du terrain, (r_b,v_b,b_b,t_b), (r_m,v_m,b_m,t_m), (r_h,v_h,b_h,t_h) sont trois couleurs associées respectivement au point le plus bas, médian et le plus haut du terrain.

Les textures sont appliquées sur un objet par :

t1d texture 2D sur des objets plans

t2d texture 2D sur un groupe d'objets, [BIER86], [SAME86].

t3d texture 3D sur un groupe d' objets.

t10d texture 2D sur un terrain, placage par rapport à la verticale.

objet3 = ?t2D(num) $U(objet1, objet2); où num est un numéro de texture correspondant à un numéro d'ordre dans une liste des textures disponibles.

Sur cette texture, on peut appliquer des transformations affines qui ne modifient pas la géométrie de l'objet.

objet_final = ~h(3) ?t2d(15) objet_base; /* le motif de la texture 2D numéro 15 sera répété 9 fois */

L'utilisation des transformation affines sur les textures est identique à celles sur les objets, seul le préfixe change : ~ au lieu de @.

Les transparences et réflexions sont indiquées par une application *aspect* notée :
^as(indice_de_l'objet, son_coefficient_de_réfraction,son_coefficient_de_réflexion)

Ces notions de transparence et de réflexion ne sont utilisées que par SKY, en effet, seul le tracé de rayons permet de reproduire ces effets optiques. Mais les objets créés peuvent être utilisés par d'autres logiciels de visualisation, développés à l'EMSE. Ce qui permet de créer et de tester rapidement la validité d'une base de données et de ne calculer une image finale en tracé de rayons que lorsque tous les paramètres (couleurs, positions ... des objets) ont été déterminés.

Mais il manque dans le langage Castor, une réelle notion de variable et les structures de contrôle qui en feraient un véritable langage de programmation. Etendre Castor pour en

faire un langage de programmation performant serait un travail de longue haleine et peut-être pas tellement utile puisque des langages de haut niveau existent déjà, avec tout l'environnement de développement approprié. C'est un langage de haut niveau à objets : C++ qui a été choisi pour modéliser les scènes. Ce n'est pas un langage interactif, mais compte tenu des temps de calcul du tracé de rayons, ce n'est pas vraiment nécessaire.

1.2 Set_up : extension C++ de Keep

C++ est un langage à objets, sur-couche de C, développé par Stroustrup [STRO86] aux ATT Bell Laboratories. Nous n'avons pas l'intention de réécrire entièrement le modeleur en C++. Nous garderons toutes les fonctions de construction d'arbres qui sont déjà écrites en C. Le but de cette extension est d'améliorer le dialogue avec l'utilisateur, lui fournir des outils plus puissants, lui apporter des moyens de contrôle lors de la création de base de données. En effet C++ est un langage fortement typé, ce qui permet d'éviter des bogues que le langage C ne saurait trouver. Par exemple, en C, employer lors d'un appel de fonction un entier au lieu d'un réel peut provoquer une erreur qui ne sautera pas aux yeux (sauf lors de l'exécution ou même de la visualisation). Mais C++, s'il est un langage à objets, ne possède pas un héritage aussi performant que des surcouches de Lisp, comme YAFOOL[2] (héritage multiple), KOOL[3], ou des langages tels SMALLTALK© ou EIFFEL© , [MEYE88]. Cette faiblesse au niveau de l'héritage a posé un certain nombre de problèmes. Nous exposerons tout d'abord les solutions choisies. Puis nous développerons les différents niveaux d'interfaces proposés. Enfin nous présenterons un exemple d'utilisation.

1.2.1 La définition d'un objet CSG

L'utilisation d'héritage en C++ pour définir un objet CSG n'est pas très simple. Prenons l'exemple suivant :

---

2 Yet Another Frame Object Oriented Langage
3 Knowledge Object Oriented Langage. © Bull
© Xerox
© Interactive Software Ingineering

```
class OBJET     {
      ~~~~~~~~;
      ~~~~~~~~;
      public:
      OBJET     une_fonction(OBJET, OBJET);
      }
class CUBE      :     OBJET     {     // CUBE est une sous-classe de OBJET
      ~~~~~~~~~;
      ~~~~~~~~~;
      }
class SPHERE    :     OBJET     {     // SPHERE est une sous-classe de OBJET
      ~~~~~~~~~;
      ~~~~~~~~~;
      }
OBJET  un_objet;
CUBE   un_cube;
SPHERE une_sphere;
un_objet.une_fonction(un_cube,une_sphere);
```

Cette dernière instruction amènera, lors de la compilation des messages et demandera de forcer la conversion de *un_cube* et *une_sphere* à OBJET pour qu'ils disparaissent. On ne peut demander à un utilisateur à chaque emploi d'une primitive de faire :

```
un_objet.une_fonction(  OBJET(un_cube),OBJET(une_sphere));
```

Les messages ne sont pas plus acceptables, ils gènent la lisibilité de la phase de compilation. Stroustrup, pour éviter ce problème, introduit la notion d'ami (friend). Ce qui transforme la déclaration de la classe objet en :

```
class OBJET     {
      ~~~~~~~~;
      ~~~~~~~~;
      public:
      friend OBJET     une_fonction(OBJET, OBJET);
      friend OBJET     une_fonction(CUBE, OBJET);
      friend OBJET     une_fonction(OBJET, CUBE);
      friend OBJET     une_fonction(SPHERE, OBJET);
      friend OBJET     une_fonction(OBJET, SPHERE);
      }
```

Ce qui devient rapidement fastidieux quand on a beaucoup de classes et de méthodes. De plus, cela oblige à modifier la classe OBJET dès que l'on veut ajouter une nouvelle primitive ce qui est plutôt contradictoire avec la philosophie de la programmation orientée objets.

Stroustrup propose une autre solution, qui consiste à définir de nouveaux constructeurs qui vont créer des variables de la classe mère. L'exemple qu'il donne est l'implantation d'une classe de nombres complexes : complex. Il utilise le fait qu'un constructeur qui n'a qu'un seul paramètre n'a pas besoin d'être appelé explicitement :

```
class   complex        {
        double reel;
        double imaginaire;
        public :
        complex(double x,double y =0.0)
        // si le deuxième paramètre est absent, c'est la valeur 0.0 qui sera utilisée
        { reel = x;      imaginaire = y; }
        complex operateur*(complex,complex);
        }

main()  {
        complex z1 = complex(23);
        complex z2 = 23;
        ~~~~~~~~~~~~~~~;
        }
```

Ces deux complexes seront créés de la même manière, par un appel à complex(double(23)), la conversion d'entier à réel est faite automatiquement. De plus, il utilise le fait qu'une constante sera interprétée comme un complexe, c'est à dire :

```
        a = b * 2;        // b est un complex
```

sera interprété comme :

```
        a = operateur*(b,complex(double(2)));
```

Mais cette solution n'est pas applicable dans notre cas où le nombre de paramètres à passer est supérieur à un. Nous avons donc choisi de ne travailler que sur un seul type OBJET. Pour créer un cube, il faut appeler une fonction qui retournera un objet de type OBJET qui sera initialisé aux valeurs d'un cube. La classe objet sera donc définie ainsi :

```
class OBJET     {
      type_description_objet  *scene;
      // c'est un pointeur sur la description  interne d'un arbre CSG
      public:
      OBJET&                  reunion     (&OBJET, &OBJET);
      OBJET&                  moins       (&OBJET,& OBJET);
      OBJET&                  inter       (&OBJET, &OBJET);
      OBJET&                  translation (&OBJET, double,double,double);
      OBJET&                  rotation    (&OBJET, double,double,double,double,
                                           double=0.0,double=0.0,double=0.0
                              // centre de la rotation, si ces valeurs ne sont pas
                              // données elles auront 0.0 pour valeur.
                                           );
      ~~~~~~;
      }
```

1.2.2 Trois niveaux d'interface

Trois niveaux d'interface ont été développés. Le niveau le plus bas correspond à l'utilisation standard C++ :

```
OBJET obj1,obj2 = CUBE(123,32,43,0), obj3 = SPHERE(213,34,255,0);
obj3  =  obj1.reunion(obj1,obj2);
```

Cette syntaxe n'est pas très agréable, elle est assez lourde et peu naturelle. Une surcouche a été définie où la méthode *reunion* a été redéfinie :

```
OBJET&     REUNION(OBJET& objet1,OBJET& objet2)
           { return obj1.reunion(obj1,obj2)}; }

OBJET obj1,obj2 = CUBE(123,32,43,0), obj3 = SPHERE(213,34,255,0);
obj3  =  REUNION(obj1,obj2);
```

Cette notation fonctionnelle est utilisée pour toutes les opérations sur les objets :

```
obj3  =  TRANSLATION(obj3,5,3,-4);
obj3  =  ROTATION(obj3,  /* axe */ 5.2,7,6,  /* angle */ 30, /* centre */ 3,6,1);
obj3  =  ROTATION(obj3 , /* axe */ 0,9,6,  /* angle */ 30 /* le centre sera par défaut (0,0,0)
*/ );
```

Cette notation est un peu verbeuse, une troisième notation a été introduite (les trois

notations sont compatibles, elles peuvent donc être utilisées en même temps). Elle se base sur la redéfinition des opérateurs, l'union se définit naturellement sous forme d'addition :

```
obj3 = obj1 + obj2;
```

Le caractère associatif est conservé :

```
obj5 = obj1 + obj2 + obj3 + CUBE(100,200,250,0);
```

Les opérateurs utilisés sont :

obj + obj	→	union
obj - obj	→	différence
obj * obj	→	intersection
obj << vect vect >> obj	→	selon le type de *vect* (ROTA,TRANS,HOMO,SYME,AFFI)[4], c'est une rotation, une translation, une homothétie ... sur *obj*.
obj <= vect vect >= obj	→	selon le type du *vect* (ROTA,TRANS,HOMO,SYME,AFFI), c'est une rotation, une translation, une homothétie ... sur la texture de *obj*.

Lors de l'exécution d'un programme de modélisation en C++, un fichier de sortie est créé, soit de type Castor, soit directement la structure de données utilisée par le programme de visualisation en tracé de rayons Yield.

Quand on applique une fonction sur un objet, par exemple TRANSLATION(obj1,1,2,3), on prend pour principe de ne pas modifier *obj1*, mais de faire une recopie de *obj1*, et c'est cette recopie que l'on modifiera. C'est le même principe que dans le langage Castor, mais on se laisse la possibilité de réaffecter un objet déjà instancié :

```
obj1 = TRANSLATION(obj1,1,2,3);
```

Pour l'instruction (A = B;), si on fait une copie bit à bit, qui est l'option par défaut en C++, cela peut poser des problèmes, puisque les pointeurs A.scène et B.scène seront identiques, et que nous voulons éviter les effets de bord. Nous redéfinissons, par conséquent, l'opérateur affectation qui fera une recopie. Mais dans ce cas l'instruction :

```
C = TRANSLATION(A,1,1,2);
```

fera deux recopies, alors qu'une seule est nécessaire. Ce qui entraînera en plus une perte de place. Nous contournons ce problème, en rajoutant à la classe objet un champ privé : **virtuel** qui aura pour valeur 0 quand l'objet a été créé par l'utilisateur et 1 quand il est issu d'une recopie. Nous appellerons virtuel tout objet *obj* tel que *obj.virtuel* soit égal à 1.

Lors d'une opération, si aucun des objets n'est virtuel on effectue une recopie, sinon un des objets virtuels est utilisé pour conserver le résultat et les objets virtuels qui ne sont plus utiles sont libérés. Lors d'une affectation, si l'opérande droit n'est pas virtuel

---

[4] ROTA, TRANS,...,AFFI sont des classes définissant des applications affines

on fera une recopie, sinon, on utilisera la place apportée par l'objet virtuel.

1.2.3 Un exemple de description de scène en C++

Nous allons présenter comme exemple la construction d'un escalier.

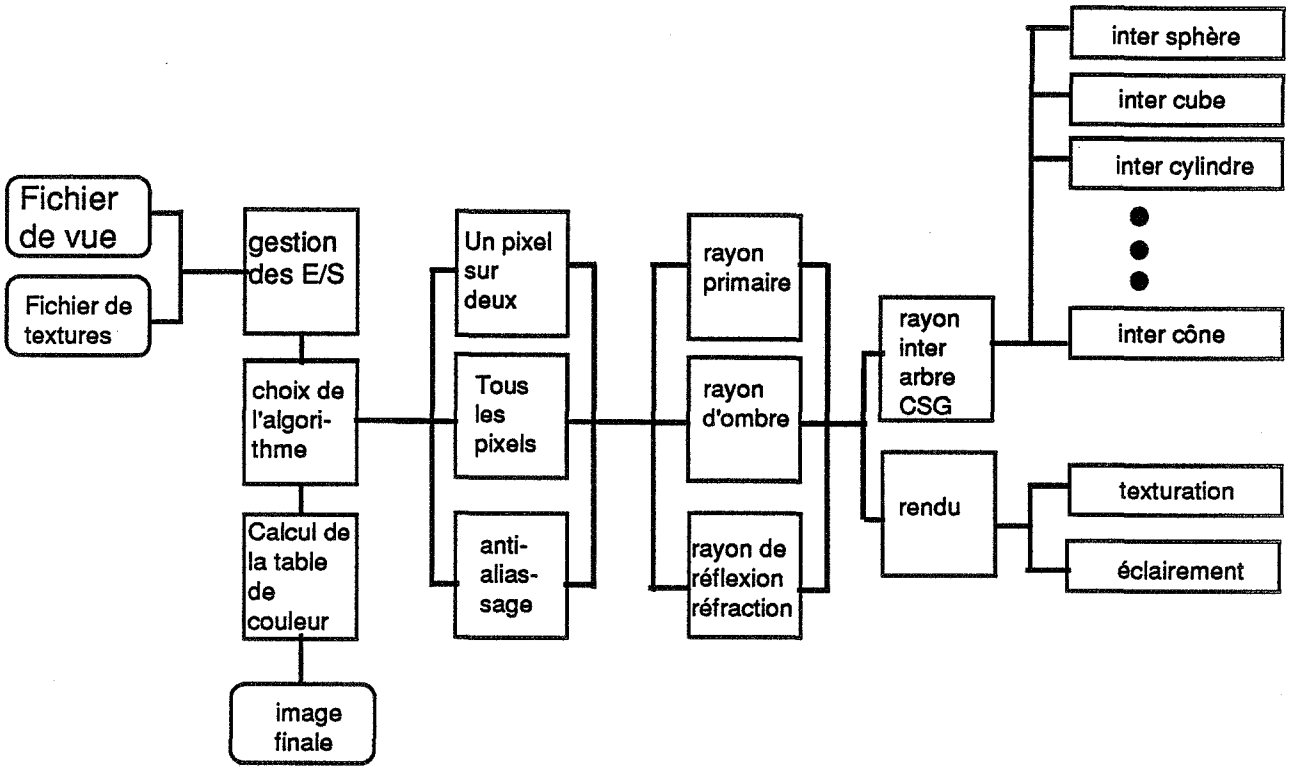
```
#include    <objet.h>
// inclusion du fichier contenant la description des types et des fonctions
// pouvant être utilisés
#define Couleur_mur 180,180,180,0

creer()  {

double  hauteur_marche      = 0.5;
double  hauteur             = 2.1;
double  angle               = 10.3;


OBJET   piece =  AFFINITE( CUBE_CREUX(Couleur_mur,0.9,0.9,0.75), 10, 10, 2.3,
                 0.5,0.5,0);
OBJET   marche  = AFFINITE(CUBE(100,100,80,0),2,1,hauteur_marche);
OBJET   escalier  = marche;

TRANS   trans(0,0,hauteur_marche);
ROTA    rota(0,0,1,angle);

while  ( trans[2] < hauteur )      {
       escalier      = escalier + (marche << rota)  <<  trans;
       rota[3]       = rota[3] + angle;
       trans[2]      = trans[2] + hauteur_marche;
       }

escalier  = NOMMER(escalier,"colimaçon");

VISU(escalier); VISU(piece);
}
```

L'utilisateur construit la scène en définissant la fonction qui s'appelle creer(). Cette fonction est ensuite compilée puis liée aux bibliothèques qui contiennent les fonctions de création des primitives. Lors de exécution du programme, on précise le nom du fichier de sauvegarde de la scène.

2 Yield : algorithme de rendu

2.1 Description générale du logiciel Yield

Nous avons déjà décrit la partie saisie des données (Set_up et Keep), nous allons rentrer un peu plus dans les détails de la partie effectuant le rendu (Yield):



La partie traitant les rayons de réflexion/réfraction a été codée par Thomas lors de son stage de fin d'étude [THOM87].

Nous allons maintenant présenter les options de visualisation.

2.2 Options de visualisation

Le programme de rendu, Yield, n'a besoin d'aucune donnée, tous les paramètres peuvent être déterminés par défaut. Ces options de visualisation peuvent se ranger dans trois catégories : la première regroupe les paramètres nécessaires à la détermination de la vue, la deuxième, les choix des techniques de rendu et la dernière, les aides pour visualiser et déboguer.

2.2.1 Options pour la détermination de la vue

Les options permettant de définir la vue sont les suivantes:

oeil *x y z* : coordonnées (*x*,*y*,*z*) de l'oeil dans le repère de la scène

visee *x y z* : coordonnées (*x*,*y*,*z*) du point regardé dans le repère de la scène

taille_image *i j* : taille de l'image finale, *i* est le nombre de lignes, *j* le nombre de colonnes

angle_vision *x y*: demi angles de vision, *x* vertical, *y* horizontal

roulis *x* : angle *x* de roulis

scene *nom* : nom du fichier contenant la description de la scène

2.2.2 Options de choix des techniques de rendu

Des options de rendu dépendent la qualité de l'image finale (et par conséquent de la rapidité du calcul de cette image). Les options sont les suivantes :

rapidite *bool* : si *bool* vaut 1 un pixel sur deux est calculé puis les pixels manquants sont obtenus par la moyenne des valeurs des voisins, si *bool* vaut 0 tous les pixels sont calculés

soleil *x y z* : coordonnées (*x*,*y*,*z*) du vecteur dirigé vers le soleil

ambiant *x* : coefficient ambiant

diffus *x*: coefficient diffus du soleil

spéculaire *x* : demande de calcul de la composante spéculaire avec un coefficient de *x*

ombre : demande de calcul d'ombres portées

reflexion *n* : demande de calcul des réflexions et réfractions jusqu'au niveau *n*

profond_quadtree n: profondeur du quadtree pour l'optimisation des réflexions/réfractions

variable : demande de calcul d'une table de couleurs variable

couleur : demande de calcul d'une image finale en 256x256x256 couleurs

debut_sp : on définit une source ponctuelle (maximum 7 sources ponctuelles)

position *xyz* : position (*x*,*y*,*z*) de la source lumineuse

portee *x*: portée de la source lumineuse

couleur *r v b:* couleur de la source lumineuse

coefficient *x y z*: coefficients diffus de cette source /* si ces coefficients ne sont par en rapport avec la couleur, cela permet de simuler plus de 7 sources */

coefficient_spec *x*: coefficient spéculaire /* utilisé avec la couleur de la source */

fin_sp : fin de la définition d'une source ponctuelle

brouillard *d c r v b* : ajout de brouillard à partir de la distance *d* suivant le coefficient *c* et de couleur (*r v b*)

fond *r v b* : couleur du fond

2.2.3 Aides

Pour un utilisateur débutant, il n'est pas toujours très facile de choisir tous les paramètres qui déterminent une vue. L'influence de ces paramètres sur une vue n'est pas toujours facile à dominer. Yield, par défaut, va choisir ces paramètres à la place de l'utilisateur. La position de l'oeil sera telle que toute la scène sera visible sur l'écran, les angles de vision seront ceux de l'oeil humain, le soleil sera positionné pour éclairer de trois quarts haut etc ... L'utilisateur peut ainsi avoir une première vision de la scène, ensuite, s'il le désire, il peut ajuster les paramètres un à un. Cette vue, étant déterminée, il peut vouloir ajuster localement sur l'image la longueur d'une ombre, l'aspect d'une texture ... Recalculer toute l'image est une perte de temps, l'utilisateur a la possibilité de ne recalculer qu'une portion de l'image en gardant les mêmes paramètres de vue. S'il veut poser par exemple un vase qu'il vient de créer sur une table, il peut récupérer, en fournissant les coordonnées d'un point (de la table) sur l'écran, la position de ce point dans le repère de la scène. Il peut aussi identifier un objet sur l'image en récupérant son nom, et ainsi le retrouver dans le fichier Castor ou C++ contenant la base de données pour modifier sa position, sa couleur ou tout autre attribut. Nous allons maintenant préciser l'emploi de ces aides :

silence : supprime les échos, c'est utile si l'utilisateur trouve le programme trop verbeux

centrer : détermine la position de l'oeil et la direction de visée afin que la scène soit centrée sur l'écran

position_bas_fenetre *i j* : position du coin inférieur gauche d'une fenêtre sur l'écran

position_haut_fenetre *i j* : position du coin supérieur droit d'une fenêtre sur l'écran

renseignement *i j* :demande de renseignements sur le point (*i*,j) de l'écran

union : supprime les différences et les intersections pour les transformer en unions, ce qui permet de visualiser la position des objets retranchés et ainsi de trouver d'éventuelles erreurs de positionnement

soleil_relatif *a h* : positionne le soleil non plus par un vecteur mais par un azimut *a* et une hauteur *h* (tous les deux en degrés) relative à la position de l'observateur et le centre de visée (cf. figure suivante)

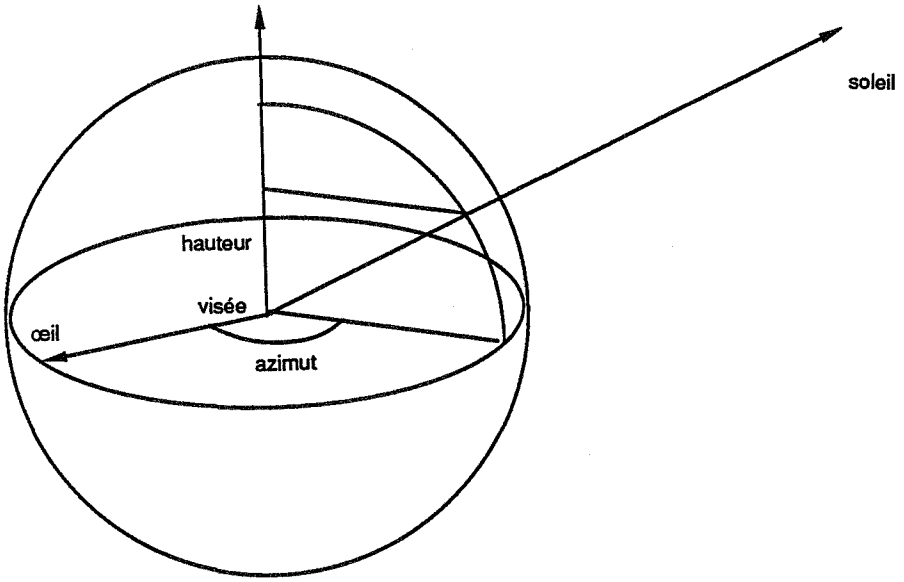


Fig. 1 Position du soleil relative à l'observateur et à la direction de visée

2.3 Visualisation d'arbres et de terrains

De nombreux systèmes de CAO proposent maintenant aux architectes et aux urbanistes des outils de modélisation d'ouvrages architecturaux (EUCLID[5], CATIA[6], ...) Des études sont menées au GAMSAU sur la paramétrisation du tissu urbain [GAMS87]. L'intérêt de cette dernière démarche est de pouvoir reconstruire l'environnement d'un objet architectural par la donnée d'un faible nombre de paramètres permettant de décrire le style d'un quartier (2-3 fenêtres marseillais, hausmanien ...). Ces deux démarches sont complémentaires, et laissent espérer à terme un outil performant pour la mise en œuvre de projets architecturaux. Mais jusqu'ici, aucun logiciel de CAO n'inclut la représentation de terrain et de sursols de type végétal. La modélisation réaliste de plantes s'est développée depuis 1982. Les résultats obtenus sont de très bonne qualité. Kawaguchi [KAWA82], Aono et Kunii [AONO84], Smith [SMIT84], Reeves [REEV85], Eyrolles, Françon et Viennot [EYRO86], [REYF88], [PRUS88], procèdent schématiquement en trois étapes :

- calcul d'une ramification par un moteur de croissance;
- plongement de cette ramification dans un espace euclidien à deux ou trois dimensions, en affectant à chaque branche de la ramification une forme géométrique et une position;
- rajout de feuilles, fleurs, fruits et rendu visuel.

Aono, Smith et Reeves utilisent une grammaire pour générer une ramification. Aono et Eyrolles se basent sur des études biologiques pour le plongement de la ramification dans $\Re^3$. Ces modèles prennent en compte non seulement les phénomènes de croissance

---

[5] Matra Datavision
[6] Dassault

des plantes et des arbres, mais aussi ceux des feuilles [LIEN87]. Ces modélisations, très précises, demandent une description fine par des surfaces gauches. Ce n'est pas l'optique que nous avons retenue. Les paysages que nous voulons visualiser ne sont pas une fin en soi, il ne sont que des faire-valoir pour l'objet architectural. Nous avons donc essayé de trouver des modèles possédant un bon rapport réalisme/temps de calcul. Pour ce faire nous modélisons le feuillage en utilisant une technique basée sur les articles de Gardner : [GARD84] et [GARD85]. Nous allons d'abord présenter la modélisation du feuillage d'un arbre [ARGE85]. Ensuite nous présenterons la méthode de représentation et de visualisation de terrain dans le cadre d'un modèle CSG.

2.3.1 Modèle spectral d'un arbre

2.3.2 Méthode de Gardner

Cette méthode permet de modéliser des arbres et des nuages à l'instar des peintres impressionnistes, en visualisant l'essence de l'objet et non pas tous ses détails.

L'analyse de Fourier permet, à partir d'une image (par exemple une saisie caméra d'un feuillage), de reconstruire la texture correspondante à partir de sinus et de cosinus. Pour simplifier, supposons que l'on possède une fonction FTEXT de $\Re^2$ dans [0..255] qui, à un point donné du plan, fait correspondre une intensité. Si l'on prend un objet E ayant la forme d'un ellipsoïde dont les axes sont parallèles aux axes de la scène, à chacun des points de E on peut associer une valeur de texture de la façon suivante :

à M(x,y,z) appartenant à E on associe $FTEXT(\arccos(\frac{x}{\sqrt{x^2+y^2}}),z)$

où arcos est la fonction réciproque de cosinus

La projection de E sur l'écran donne un contour C dont l'équation (F(X,Y) = 0) est facile à calculer connaissant l'équation de l'ellipsoïde E et le plan de l'écran. La fonction F est une fonction dont le maximum MAX est atteint au point de projection sur l'écran de l'origine des axes de l'ellipsoïde, qui s'annule sur le contour de C et qui est toujours négative à l'extérieur du contour. Donc la fonction $\frac{F(X,Y)}{MAX}$ est maximum à l'intérieur du contour, de maximum 1, et négative à l'extérieur de C.

A l'aide de la valeur de FTEXT en un point M de E à afficher, de la valeur de $\frac{F}{MAX}$ en la projection de M sur l'écran, on détermine si un point est visible ou non en utilisant un seuil minimal. Dans le cadre d'un algorithme de tracé de rayons nous utilisons une méthode différente basée sur les propriétés des rayons.

2.3.2.1 Intégration de la méthode spectrale dans le tracé de rayons.

Nous travaillons dans le repère local d'un ellipsoïde. Ainsi nous sommes ramenés à un repère dans lequel l'ellipsoïde est une sphère centrée en (0,0,0) et de rayon 1.

La texture est déterminée par la formule suivante:

$$FTEXT(X,Y,Z) = \sum_{i=1}^{i=n} c_i(\sin(\omega_i X + \sin(\omega_i' Y)) + 1) * \sum_{i=1}^{i=n} c_i(\sin(\omega_i Y + \sin(\omega_i' X)) + 1)$$

où n, c_i, ω_i, ω_i' sont des constantes et $X = \frac{x}{\sqrt{(x^2+y^2)}}$ et $Y = Z$.

C'est une expression simple, Schachter [SCHA80] a exposé des formules plus générales. On remarque que la formule est périodique, ce qui permet de ne pas avoir de problèmes de continuité entre X = -1 et X = 0. Cette texture sert non seulement à déterminer la couleur du feuillage mais aussi à déterminer la découpe de sa silhouette.

La procédure d'intersection entre une sphère et un rayon[7] calcule la distance D entre le rayon et le centre de la sphère. S'il y a intersection, (1-D) est comprise entre 1 et 0. Cette valeur correspond à la fonction $\frac{F}{MAX}$, définie dans le paragraphe précédent. Nous utilisons deux seuils : SEUIL1 et SEUIL2. Notons C la couleur du point d'intersection, calculée à partir de FTEXT. Si C * (1-D) est inférieur à SEUIL1 on élimine ce point d'intersection. Si (C * (1-D)) est compris entre SEUIL1 et SEUIL2, on calcule un coefficient de transparence qui varie de 0 a 1 entre SEUIL1 et SEUIL2. Si (C * (1-D)) est supérieur à SEUIL2 on garde le point d'intersection sans aucun changement.

[7] cf chapitre 1

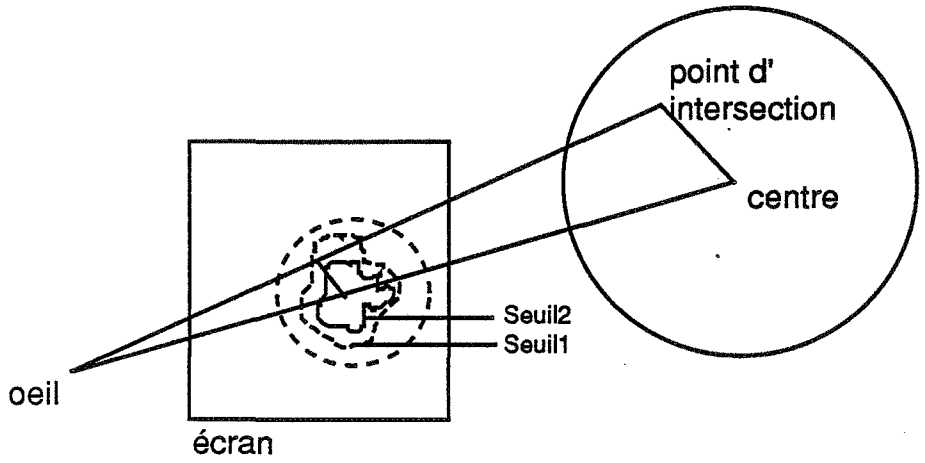


Fig.2 Détermination de la silhouette.

En réalité c'est une fonction de $\frac{1}{\sqrt{D}}$ - 1, au lieu de (1-D), que l'on utilise pour avoir un découpage plus naturel. On utilise de plus une texture sur les normales, et non plus seulement sur la couleur, pour donner plus de relief au feuillage ([BLIN78]).

Cette même technique peut être utilisée pour représenter des nuages, en prenant une texture moins heurtée et en augmentant la zone rendue transparente.

2.3.2.2 Terrains

Nous travaillons sur des terrains donnés par une grille altimétrique. Chaque maille d'une grille rectangulaire possède une altitude. Bien évidemment, avec ce type de représentation on ne peut pas avoir de surplomb. Mais si l'on considère que ce terrain est un volume déterminé par la surface du terrain et limité par un plan horizontal et par les quatre plans verticaux qui délimitent la grille, ce volume peut être utilisé dans un arbre de construction. Ce qui permet de creuser des routes, des fondations de constructions, des tunnels, des carrières etc .., et par conséquent autorise les surplombs et les excavations.

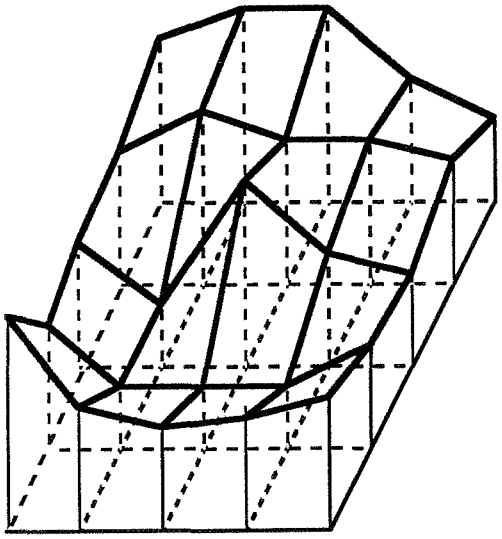


Fig. 3 Grille altimétrique

Nous rappelons l'algorithme utilisé par Coquillart [COQU84]. L'algorithme n'effectue que O(N) tests d'intersection avec une facette, où N^2 est le nombre de facettes : Considérons la projection du rayon sur le plan de la grille. Il est évident que le rayon ne peut intersecter que les facettes des cases traversées par la projection du rayon; il suffit donc de tester ces facettes; les cases empruntées par le rayon en projection sont déterminées par une variante de l'algorithme de BRESENHAM. Cet algorithme peut être employé pour tous les types de rayons : primaires et secondaires. Si on utilise cet algorithme dans un arbre de construction, il faut, pour chaque point de l'écran sur lequel se projette le terrain faire exactement N tests si l'oeil ne se trouve pas au dessus de la grille et en moyenne N/2 s'il s'y trouve, pour les rayons primaires, N/2 pour chaque rayon d'ombre, réfléchi ou réfracté. En général, on n'a besoin que du premier point d'intersection pour les rayons primaires et d'ombre, et il n'y a que peu de rayons de réflexion-réfraction. Si l'on considère uniquement les rayons primaires et le premier point d'intersection, on peut faire de substantielles optimisations, en utilisant la cohérence verticale, mais qui sont bien plus complexes [COQU84]. Nous proposons d'utiliser un algorithme simple pour trouver le premier point d'intersection pour les rayons primaires. Cet algorithme est une extension de l'algorithme de Anderson [ANDE82] à un quadtree. Dans le cas où il faut plus d'un point d'intersection, on utilisera une variante de

l'algorithme de parcourt d'un quadtree de Kunii et Wyvill [KUNI85].

2.3.2.2.1 Principe de l'algorithme de Anderson

Anderson détermine l'ordre d'affichage des facettes, par rapport à la position de l'oeil. En utilisant le fait que, si la facette F_1 a un ordre supérieur à F_2, alors F_2 ne peut pas cacher F_1. Sur la figure suivante sont décrits les trois cas de figure et les ordres d'affichage. Anderson garde un périmètre d'affichage, si une facette se projette à l'extérieur de ce périmètre, la partie de la facette qui dépasse du périmètre est vue, on l'affiche donc et le périmètre est augmenté de cette partie. Pour stocker ce périmètre Anderson utilise une méthode un peu compliquée, nous utiliserons une méthode simplifiée ainsi qu'un ordre d'affichage légèrement différent.

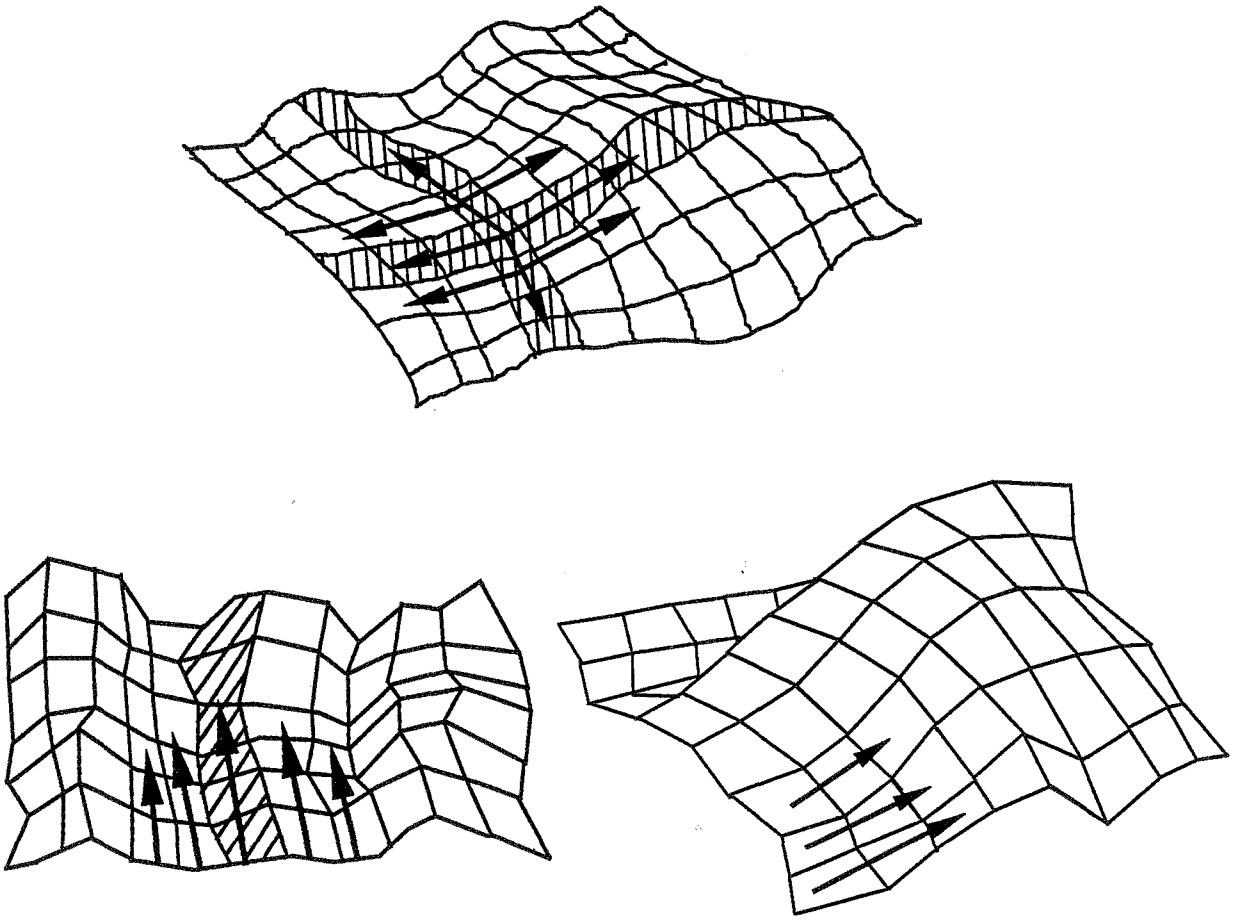


Fig.4 Ordre d'affichage dans les trois cas de figure

2.3.2.2.2 Méthode d'affichage d'un terrain stocké sous forme de quadtree

Nous possédons la description d'un terrain sous forme d'une grille altimétrique, à partir de cette grille, un quadtree est déduit de la manière suivante : une hiérarchie des boîtes englobantes est formées en utilisant la boîte englobante associée à chaque facette. Au niveau 0, une seule boîte englobe entièrement le terrain, au niveau 1, 4 boîtes englobent chacune un quart du terrain etc ...(cf figure suivante). Un premier algorithme de visualisation permet de trouver pour chaque pixel quelle est la facette vue. Pour plus de rapidité, cette visualisation dépend d'une profondeur maximale dans le quadtree, ce qui permet de visualiser le terrain avec différents niveaux de précision. De plus, l'utilisateur est laissé libre de définir une taille minimum sur l'écran, en dessous de laquelle le quadtree ne sera plus subdivisé, ainsi la définition du terrain est liée à sa taille sur l'écran.

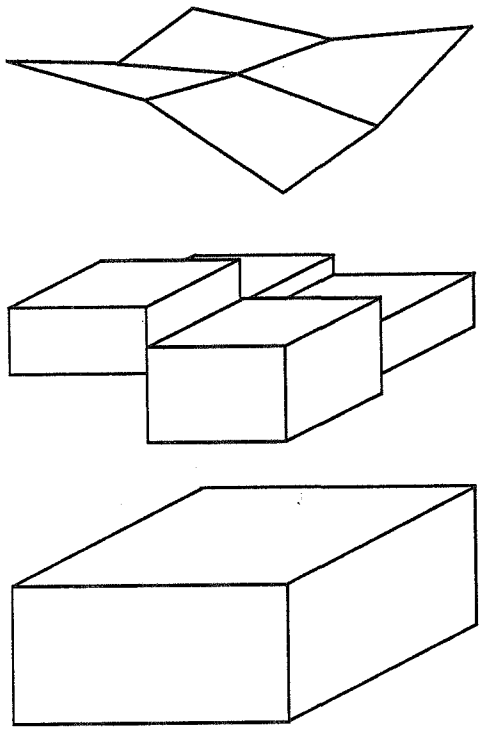


Fig. 5 Hiérarchie des boîtes englobantes

Pour stocker la projection des facettes du terrain, nous utilisons un tableau, *sland*, d'entiers dont la taille est égale à celle de la projection de la boîte englobante du terrain sur l'écran. A chaque point du tableau correspond un pixel. L'entier contenu dans le tableau désignera une facette, qui s'y projette, déterminée par sa position (i,j) et sa profondeur *decal* dans le quadtree associé au terrain ; si aucune facette ne s'y projette l'entier vaudra -1.

Nous allons maintenant décrire plus précisément l'algorithme d'affichage. Nous partons de tableau *sland* initialisé à -1. Nous allons présenter l'algorithme récursif d'affichage du quadtree sur le tableau. Soient (x_o,y_o,z_o) les coordonnées de l'oeil dans le repère de la scène. Soit *taille_min* la taille minimum d'une facette sur l'écran. Soit *quad_land* le quadtree associé au terrain Soit *prof_max*, la profondeur maximum dans le quadtree (cette profondeur donnée par l'utilisateur est inférieure à la profondeur maximum réelle du quadtree : *prof_réelle*). Soit *decal* la profondeur courante dans le quadtree (au départ *decal* vaut 0). Soient (i,j) les coordonnées de la cellule courante du quadtree. L'algorithme peut s'écrire de la manière suivante :

```
Projection_terrain(xo,yo,zo,prof_max,decal,i,j,sland)
{
si  decal ≥ prof_max ou
    projection(boîte_englobante(quad_land,i,j,decal)) est égale à un nombre de -1,
          dans sland, inférieur à  taille_min
    afficher(sland,facette(quad_land,i,j,decal));
sinon
    dep = 1 << (prof_réelle - decal)
    si  ((xo≤ i + dep) et (yo≤ i + dep)) alors      /* cas 1*/
        Projection_terrain(xo,yo,zo,prof_max,decal+1,  i       , j       ,sland)
        Projection_terrain(xo,yo,zo,prof_max,decal+1,  i+dep   , j       ,sland)
        Projection_terrain(xo,yo,zo,prof_max,decal+1,  i       , j+dep   ,sland)
        Projection_terrain(xo,yo,zo,prof_max,decal+1,  i+dep   , j+dep   ,sland)
    fsi
    si  ((xo≤ i + dep) et (yo> i + dep)) alors      /* cas 2 */
        Projection_terrain(xo,yo,zo,prof_max,decal+1,  i       , j+dep   ,sland)
        Projection_terrain(xo,yo,zo,prof_max,decal+1,  i+dep   , j+dep   ,sland)
        Projection_terrain(xo,yo,zo,prof_max,decal+1,  i       , j       ,sland)
        Projection_terrain(xo,yo,zo,prof_max,decal+1,  i+dep   , j       ,sland)
    fsi
    si  ((xo> i + dep) et (yo≤ i + dep)) alors      /* cas 3 */
        Projection_terrain(xo,yo,zo,prof_max,decal+1,  i+dep   , j       ,sland)
        Projection_terrain(xo,yo,zo,prof_max,decal+1,  i+dep   , j+dep   ,sland)
        Projection_terrain(xo,yo,zo,prof_max,decal+1,  i       , j       ,sland)
        Projection_terrain(xo,yo,zo,prof_max,decal+1,  i       , j+dep   ,sland)
    fsi
    si  ((xo≤ i + dep) et (yo≤ i + dep)) alors      /* cas 4 */
        Projection_terrain(xo,yo,zo,prof_max,decal+1,  i+dep   , j+dep   ,sland)
        Projection_terrain(xo,yo,zo,prof_max,decal+1,  i       , j+dep   ,sland)
        Projection_terrain(xo,yo,zo,prof_max,decal+1,  i+dep   , j       ,sland)
        Projection_terrain(xo,yo,zo,prof_max,decal+1,  i       , j       ,sland)
    fsi
fsi
}
```

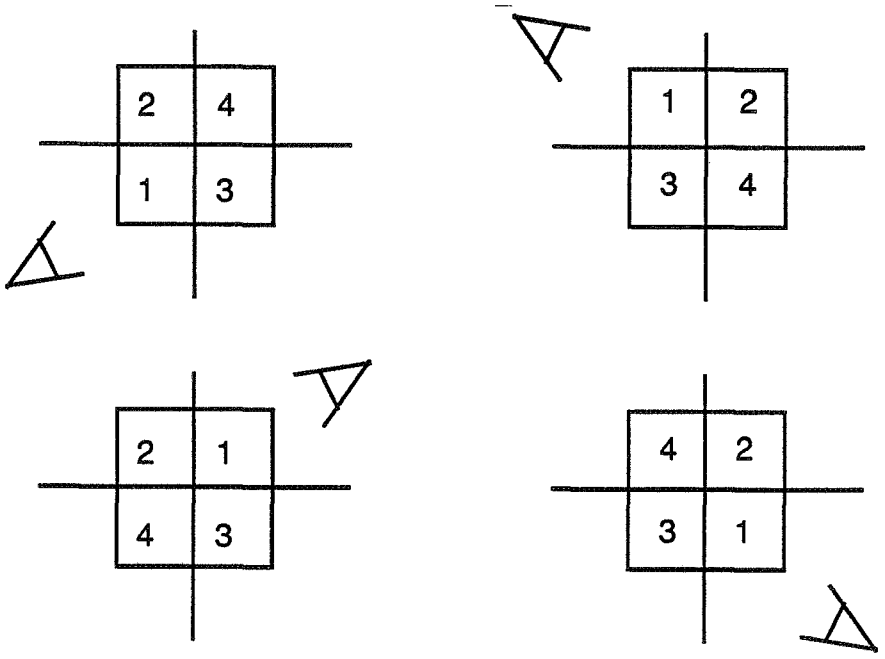


Fig.6 Ordre de parcours récursif du quadtree

Pour un rayon primaire (i,j), il suffit de chercher dans le tableau la facette associée au pixel (i,j), puis on calcule l'intersection entre le rayon et cette facette, soit en considérant que cette facette est un patch bilinéaire soit que cette facette est composée de deux triangles et en opérant un lissage bilinéaire sur les normales, ce qui donne, très rapidement, un bon résultat. Ce premier point d'intersection suffit si le terrain n'intervient pas dans une opération différence ou intersection. Sinon à partir de ce point, on cherche d'autres intersections éventuelles en suivant la projection du rayon sur le sol. Pour un rayon secondaire, nous utiliserons aussi, la projection du rayon sur le sol, ainsi que l'algorithme de passage d'une cellule d'un quadtree à la suivante, présenté dans le Chapitre 2, paragraphe 5.2.3. Avant de tester l'intersection d'une facette F et d'un rayon, nous pouvons tester l'intersection entre le rayon et toutes les boîtes qui englobent F. Si l'une de ces boîtes n'est pas intersectée, le rayon n'intersecte pas F, nous pouvons passer à la facette suivante. Mais ceci peut être encore amélioré : nous n'allons pas tester toute la hiérarchie des boîtes qui englobent la facette F. En effet, soit F' la facette précédemment testée. Soit B_i la plus grande boîte qui englobe à la fois F et F'. Cette boîte a déjà été testée, il est inutile de recommencer un test. Il suffit donc de retrouver la plus grande boîte B_i contenant F et F'(ce qui est trivial), et de ne tester uniquement les boîtes contenant F et incluses dans B_i. Maintenant, si on suppose qu'une boîte B_j n'est pas intersectée par le rayon. Toutes les facettes contenues dans B_j ne seront pas intersectées. On peut passer à la cellule suivant toutes celles contenues dans B_j (cf figure suivante).

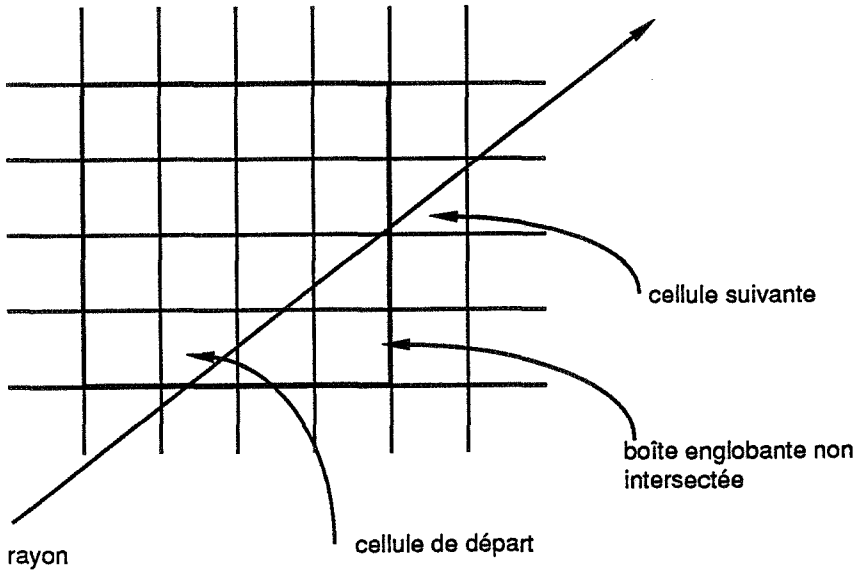


Fig.7 Test des boîtes englobantes, passage à la cellule suivante.

L'optimisation que nous avons présenté permet d'obtenir pour un rayon primaire dans le cas d'unions, la facette intersectée, en $O(1)$, au lieu de $O(N)$, comme c'est le cas de l'algorithme de Coquillart. Le temps de précalcul n'est pas très élevé par rapport au temps de visualisation du terrain.

2.4 Textures

La présence de textures apporte, à moindre frais, un très grand réalisme à une image synthétique. Elles permettent de rendre les détails du matériau composant une surface, qui, sinon, présenterait un aspect trop lisse. On peut distinguer les textures macroscopiques (briques d'un mur) des textures microscopiques (granularité de la brique) en prenant en compte la notion de distance d'observation. On peut distinguer, en plus, deux classes de textures :

- 2D : c'est une image (une tapisserie) plaquée sur une surface (comme un papier peint sur un mur),
- 3D : c'est un matériau,défini dans tout l'espace, que l'on sculpte suivant les contours d'un objet.

Les méthodes de génération de texture peuvent être divisées en six classes [PERO87], [MAGN87]:

1 méthodes fractales

2 méthodes spectrales

3 méthodes syntaxiques

4 méthodes structurelles

5 méthodes statistiques

6 méthodes de saisie par caméra ou par palette

Nous nous limiterons aux textures 2D. Nous supposerons que ces textures ont été générées par l'une des six méthodes précédentes et tabulées pour diminuer les temps de calcul d'une image en tracé de rayons. Nous voulons stocker ces textures en respectant au mieux les conditions (hélas contradictoires) suivantes :

- occuper le moins de place possible
- prendre le moins de temps possible lors de l'antialiassage

Pour ce faire nous calculons toutes les textures à haute résolution, puis nous filtrons pour nous retrouver à une résolution 256x256 (qui semble une définition suffisante). Pour antialiasser une texture, nous voulons utiliser la méthode de Williams [WILL83] qui utilise une pyramide dont chaque niveau correspond à un certain taux de compression de l'image. Stocker entièrement cette pyramide demanderait :
(256*256+128*128+64*64+32*32+16*16+8*8+4*4+2*2+1)*3 = 253 743 octets, soit 247 ko. Nous économisons à peu près un quart de la place nécessaire en ne sauvegardant que (256*256+32*32+16*16+8*8+4*4+2*2+1)*3 = 196 ko, stockés en suivant l'ordre de Peano[8]. Pour obtenir les valeurs stockées au niveau 128*128, qui sont le mélange de quatre valeurs du niveau 256*256, nous utilisons le fait que les quatre couleurs que nous devons obtenir pour faire le mélange, sont consécutives suivant l'ordre de Peano ; nous pouvons donc rapidement faire le mélange en accédant à la première valeur, puis en ajoutant les trois couleurs qui suivent[9]. De même, au niveau 64*64, nous n'avons qu'à chercher la première couleur puis à ajouter les 15 couleurs suivantes. Pour ne pas perdre trop de temps dans les calculs de mélange, nous avons stocké les niveaux 32*32,16*16 ... 1, car nous aurions, sinon, à faire de 64 à 256*256 additions pour retrouver ces niveaux. Pour trouver la couleur c associée au point (x,y), connaissant le taux de compression tx, nous calculons :

soit $t_{i-1} < tx \leq t_i$, où i-1 et i sont deux niveaux consécutifs dans la pyramide, nous

[8] cf chapitre 4

[9] ce qui se fait de façon très performante en C

allons faire deux interpolations :

c_{i-1} = interpolation entre les quatre couleurs, du niveau i-1, qui environnent (x,y)

c_i = interpolation entre les quatre couleurs, du niveau i, qui environnent (x,y)

$$c = \frac{(tx - t_{i-1})}{(t_i - t_{i-1})} c_i + \frac{(t_i - tx)}{(t_i - t_{i-1})} c_{i-1}$$

3 Parallélisation de Yield

3.1 Description de la machine hôte : CAPITAN

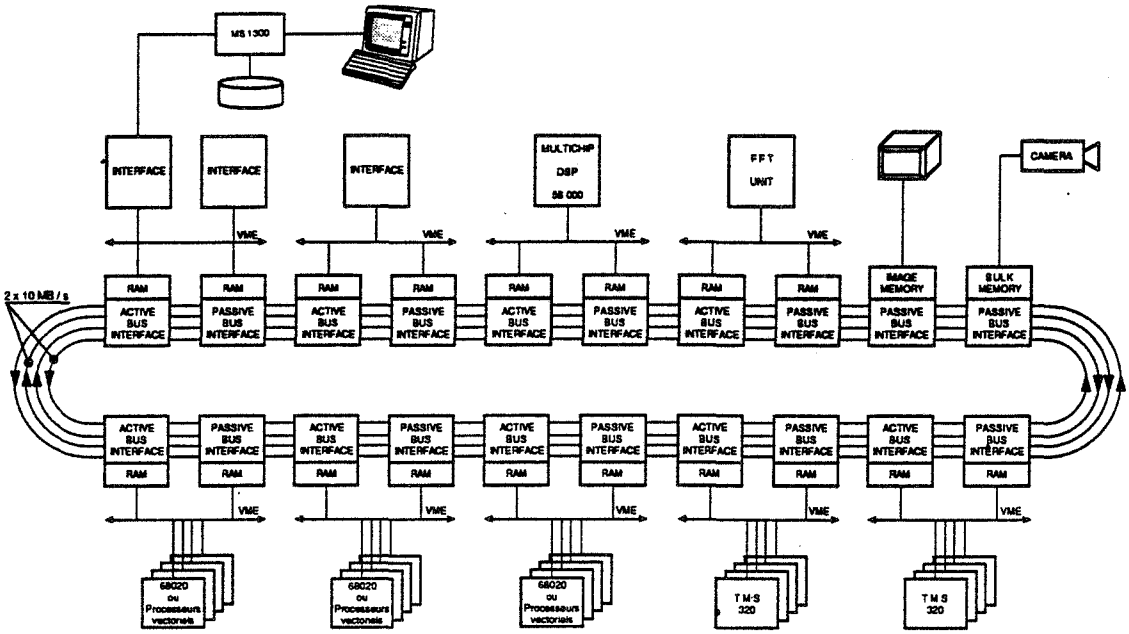
3.1.1 Description matérielle

Capitan est un calculateur paralléle de type MIMD (Multiple Instruction Multiple Data) constitué de noeuds au standard VME reliés par un bus en anneau.

Un nœud peut comporter

- des cartes microprocesseur standard : 68020, 80386, 68030.
- des processeurs de traitement du signal, TMS320, DSP56000.
- des processeurs vectoriels
- des cartes mémoire de masse
- des cartes mémoires d'image reliées à un moniteur ou une caméra
- des cartes de liaison avec un autre ordinateur ou un réseau Ethernet

En fait, chaque nœud est construit autour d'un bus local au standard VME. Ce qui permet de connecter tout autre matériel au standard VME. Un réseau de communication relie les différents nœuds de Capitan. Il est formé de deux bus en anneau qui ont chacun un débit de 10 Mo/s, ce qui autorise un débit de 20 Mo/s. Chaque bus peut être reconfiguré, par logiciel et indépendamment de l'autre, en plusieurs sous-bus qui travailleront sur un sous-ensemble de nœuds. Ainsi, on peut travailler sur des structures en forme de pipeline ou de tableau bidimensionnel de processeurs ("2D array processor"). La configuration type est la suivante :



3.1.2 Description logicielle

Un logiciel tournant sur Capitan est composé de deux parties. La première partie est un ensemble de tâches séquentielles qui communiquent entre elles. Chaque tâche a été décrite de façon séquentielle en C, FORTRAN ou assembleur. Des échanges de messages permettent aux tâches de communiquer entre elles. La deuxième partie est une description de la configuration de la machine, la distribution des différentes tâches sur les processeurs et les messages échangés entre les tâches. Plusieurs tâches peuvent se trouver sur un même processeur, car il existe un système d'exploitation multitâches. D'un autre côté, une tâche peut être dupliquée sur un ou plusieurs processeurs.

Les tâches dialoguent entre elles par l'intermédaire de primitives qui s'emploient sous forme de procédure :

envoi d'un message
- send : envoi d'un message à une tâche donnée
- brcast : envoi d'un message à toutes les tâches

réception d'un message
- sroe : attente d'un message provenant d'une tâche donnée
- receiv : attente d'un message provenant de n'importe quelle tâche
- tnrec : test, sans blocage, d'un message émis par n'importe quelle tâche

Il existe aussi des procédures pour écrire ou lire dans la mémoire de masse, si elle existe dans la configuration.

De plus, il existe, actuellement, un debogueur symbolique tournant sur le nœud UNIX, qui permet de tester l'exécution de chaque tâche par un appel de dbxtool (débogueur symbolique dans l'environnement Suntools) sur chacune de ces tâches et qui visualise les échanges de message. Ce débogueur travaille en réalité en séquentiel (les bogues provenant de l'échange de message asynchrone ne peuvent pas toujours être détectés), il est donc relativement lent, mais il peut servir de simulateur de Capitan. Un débogueur symbolique tournant directement sur Capitan est en cours de développement. On peut, lorsqu'un programme a échoué, récupérer sur Capitan le type d'erreur et les messages échangés

3.1.3 Description de la configuration utilisée

Par logiciel, en utilisant LSL, le langage de spécification de liens, nous avons défini les nœuds et les processeurs utilisés sur chaque nœud. Nous nous sommes limités à un nœud Unix comportant la station Matra MS 3/110 C, cinq processeurs sur le nœud PN1, quatre sur le nœud PN4. Ceci pour nous restreindre à des processeurs identiques et ainsi garder une certaine homogénéité pour déterminer les temps de calcul. Nous avons donc ainsi, neuf processeurs 68020 avec une mémoire de 1 Mo et un coprocesseur de virgule flottante.

3.2 Les types de parallélisations que rend possibles CAPITAN

Capitan permet d'utiliser presque directement un logiciel écrit en séquentiel et de distribuer des parties du code sur les différents processeurs. Il n'est pas nécessaire de le réécrire dans un langage parallèle. Sans parler des architectures dédiées, on peut envisager deux types de parallélisation. La première, sur les pixels, ne demande pas de réécrire le logiciel, il suffit, en gros de le distribuer sur chacun des processeurs puis de répartir les pixels à calculer, ensuite on récupère la couleur finale du pixel. Cela demande aussi de recopier sur chaque processeur la base de données. Ce qui limite la taille de la base de données (ou alors il faut beaucoup de mémoire sur chaque processeur, quelques dizaines de Giga-octets pourraient suffire). Par contre le volume des échanges de message est assez faible (distribution de la base de données, des données déterminant la vue, pour chaque pixel : le numéro du pixel à calculer puis le numéro du pixel calculé et sa couleur).

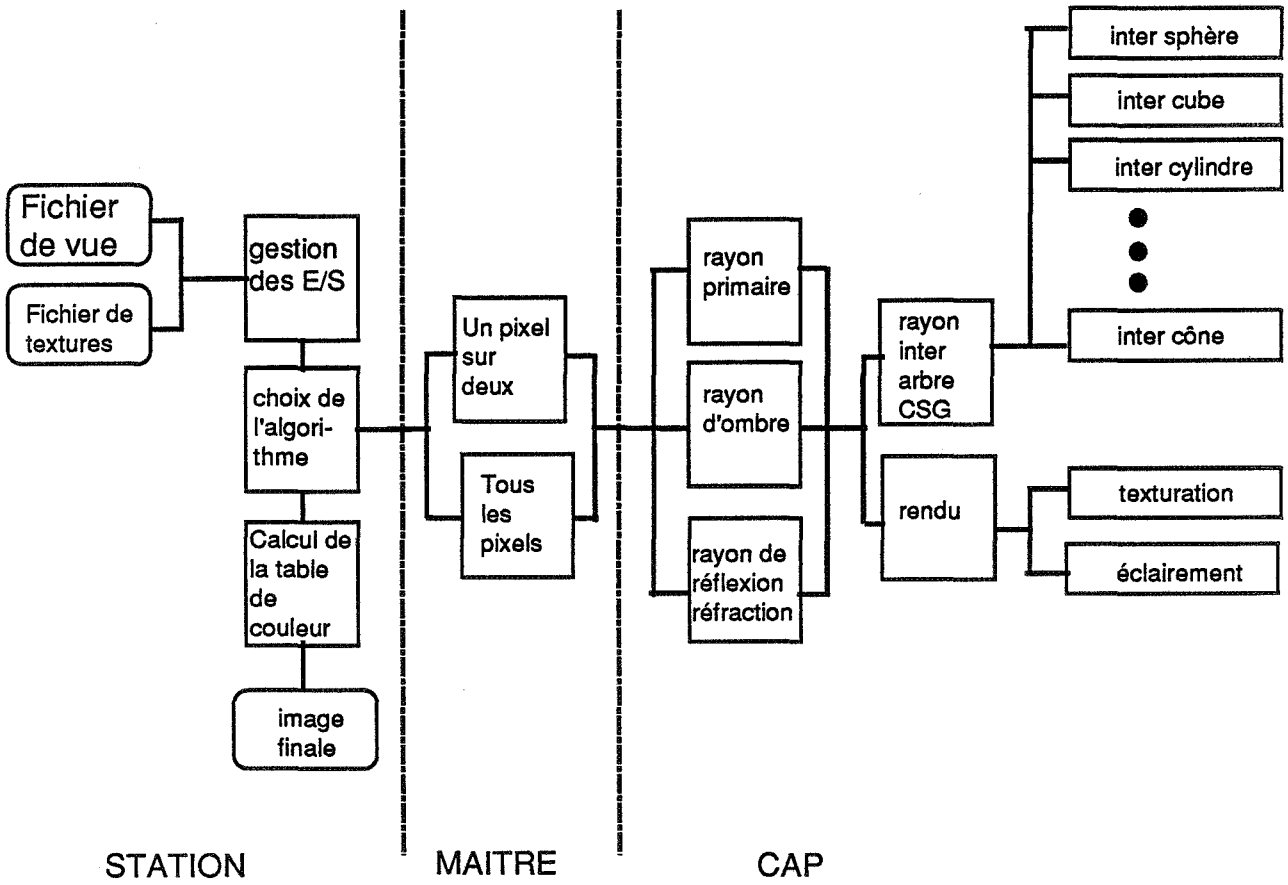
La deuxième méthode distribue la base de données et non plus les pixels. Elle permet d'augmenter le nombre d'objets pouvant être visualisés. La taille limite existe toujours, mais elle est multipliée par le nombre de processeurs. Par contre, la distribution n'est pas un problème simple si on veut éviter à des processeurs d'être sous-utilisés. Un processeur sur-utilisé peut amener une situation de blocage. Par exemple, si la zone de l'espace objet contenant une source lumineuse est allouée à un processeur P, P peut avoir à tester tous les rayons d'ombre produit par cette source. Le volume des échanges de message, par rapport à la méthode précédente, est largement augmenté. Ainsi, on peut être amené à optimiser non plus le temps de calcul par processeur mais le nombre de messages échangés.

3.3 Un essai de parallélisation de Yield

Nous choisissons de distribuer les pixels aux processeurs. Nous sommes ainsi assurés que tous les processeurs auront une charge équivalente. Cette méthode nous permet aussi de garder telles quelles les optimisations déjà étudiées dans le cas séquentiel. Sur chaque processeur, nous avons des structures de données pour les rayons primaires et secondaires, mais les objets ne sont pas vraiment dupliqués seul les pointeurs le sont, elles ne sont donc pas énormes. Seul le terrain qui est un objet particulièrement volumineux ne sera pas, pour le moment, utilisé. La méthode d'antialiassage que nous avons présentée dans le chapitre 3, demande une connaissance locale (3x3 pixels) de l'image, elle ne peut donc s'appliquer directement puisque lors du calcul en parallèle, tous les pixels sont indépendants. En réalité, on distribue non pas des pixels mais des segments de pixels ce qui permet d'économiser du temps puisque pour tous les pixels d'un segment donné, la sous-scène, pour les rayons primaires, est la même. On pourrait donc utiliser une connaisance sur un segment pour antialiasser. Mais nous nous bornerons à une visualisation non antialiasée pour tester les performances de notre algorithme de tracé de rayons sur Capitan.

3.3.1 Description des différentes taches

Nous allons découper Yield en trois tâches distinctes, Station, Maitre, Cap. Si nous reprenons le graphique représentant Yield, le découpage se fera naturellement :



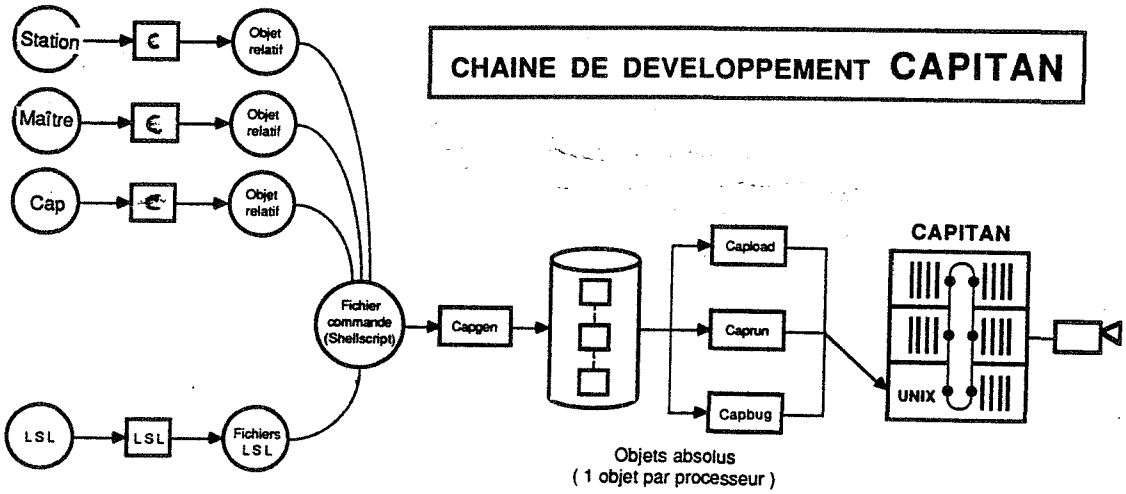
Les fonctionnalités des différentes tâches sont les suivantes :

Station : distribution des données de la vue, de la base de données, récupération de la couleur des pixels calculés;

Maitre : réception des données de la vue et de la scène
boucle de distribution des segments de pixels à calculer

Cap : réception des données de la vue et de la scène
boucle de réception d'un segment à calculer
calcul de tous les pixels de ce segment
envoi à Station des couleurs de ces pixels

La chaîne de production de l'exécutable est la suivante :

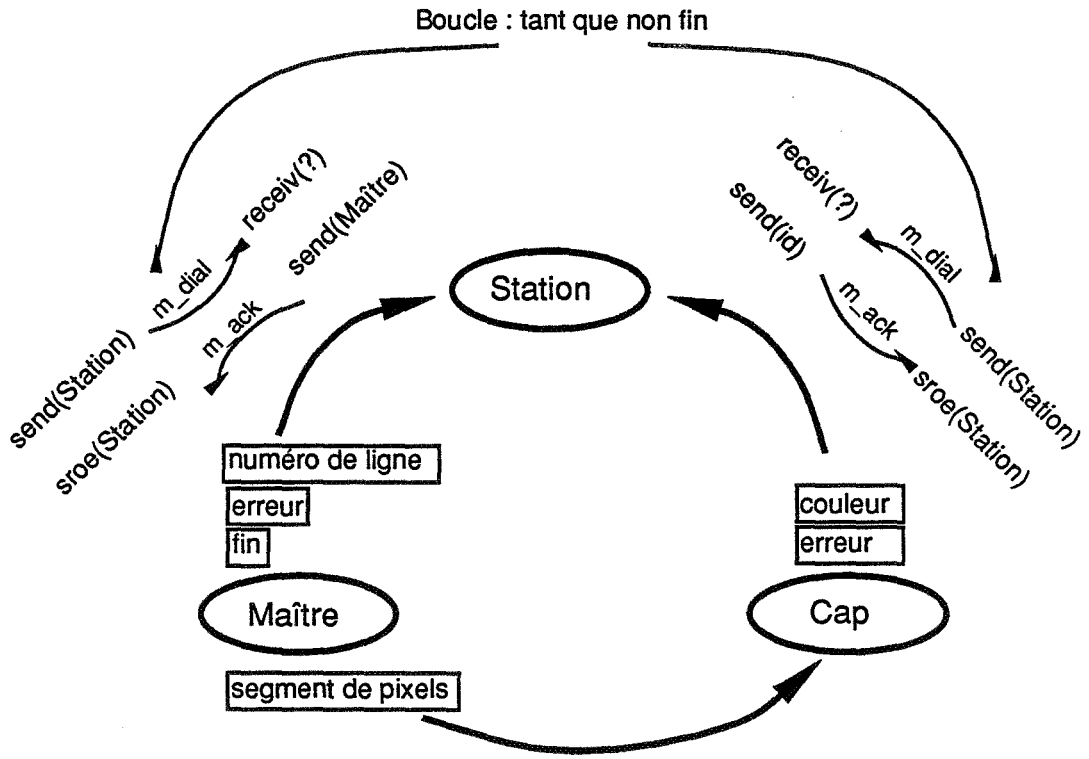


LSL : Compilateur LSL
Capgen : Génération des objets absolus (1 objet par processeur)
Capload : Loader Capitan
Capbug : Debugger Capitan

Nous allons maintenant détailler les différents échanges de messages.

3.3.2 Echanges de messages

Le programme se découpe en quatre stades d'échange de message. La première étape est nécessaire pour compter le nombre de processeurs disponibles, et éventuellement se limiter, de façon logicielle à un certain nombre de processeurs. Nous pouvons ainsi établir facilement un graphique prennant en compte les temps de calculs par nombre de processeurs. Ces résultats seront exposés en annexe. Le deuxième stade recouvre toute la distribution des données aux processeurs. La troisième partie est le corps principal du programme : le calcul des pixels. Enfin lors de la dernière étape on détectera la fin des calculs.



id désigne une tâche de type CAP

types de message: m_ack et m_dial

types de renseignements transités par m_ack :

accord : 0 entier (la simple existence de ce message suffit)

types de renseignements transités par m_dial :

numéro de ligne : 1 entier
erreur : 1 entier (code d'erreur)
fin : 1 entier (code de fin)
segment de pixels : 2 entiers (numéro de ligne numéro, de segment dans la ligne)
couleur : 5 entiers (numéro de pixel, couleur)

4 Conclusion

Sur une scène test comportant une quarantaine de primitives, nous passons d'un temps de 337.7 secondes[10] à 52.76 en utilisant les optimisations logicielles présentées dans les chapitres précédents. Après parallélisation, cette image est calculée en 6.4 secondes, ce qui donne une optimisation linéaire en nombre de tâches CAP. Au total nous obtenons un gain de temps d'environ 53 fois.

[10] sur une station Matra MS 1300 (équivalente à un Sun 3/110)

Conclusion

Plusieurs algorithmes originaux utilisant les arbres de construction pour modéliser les scènes ont été exposés dans cette thèse. Ces algorithmes ont été mis en œuvre au sein du logiciel de tracé de rayons SKY. Ce logiciel calcule des images en rouge, vert et bleu sur 256 niveaux pour chacune des composantes. Ces images peuvent être visualisées sur toute mémoire d'image par un algorithme original qui calcule une table de couleurs variable en utilisant une quantification des couleurs par un octree en suivant le balayage de Peano. Ces images sont antialiassées par une nouvelle méthode d'antialiassage détectant les petits objets, elle prend en compte les information géométriques retournées par le tracé de rayons. Elle reconnaît ainsi les zones présentant de l'aliassage. Cette méthode peut être utilisée dans le cadre d'un tracé de rayons distribué afin de déterminer le nombre de rayons à relancer selon les zones.

Une solution simple permettant de résoudre de façon simple et générale les ambiguïtés du modèle CSG a été étudiée. Un nouveau type de primitive : les primitives creuses, a été défini pour encore améliorer les performances du tracé de rayons.

Nous avons proposé une technique réalisant les opérations booléennes directement sur les boîtes englobantes. Cette méthode optimale peut être utilisée non seulement en deux dimensions sur la projection des boîtes englobantes sur l'écran, mais aussi de façon immédiate en trois dimensions. Elle peut donc être utilisée avec d'autres méthodes d'optimisation. Le choix de travailler sur une projection sur l'écran a été fait dans le but (non encore avoué) d'utiliser le tracé de rayons pour construire ("interactivement") des scènes. En effet, si on rajoute une primitive dans une scène dont une image a déjà été calculée, seuls les pixels compris dans le rectangle englobant associé à cette nouvelle primitive doivent être recalculés. Le tracé de rayons permet très simplement de désigner un objet sur l'écran et de le retrouver dans l'arbre CSG; on peut vouloir ainsi, par exemple, le déplacer. De la même façon que précédemment, seuls les pixels contenus dans le rectangle courant et le précedent doivent être réévalués. On devrait ainsi (selon le type de matériel utilisé) obtenir une méthode rapide, économique en place mémoire, de construction interactive de scène.

Le logiciel de tracé de rayons SKY est actuellement utilisé au sein de l'équipe communications visuelles de l'EMSE en vue de lui adjoindre des effets lumineux plus complexes tels que des cônes (ou des cylindres) de lumière. SKY a été porté sur un HP9000, un SPS90, un SUN3/110 et sur une machine parallèle CAPITAN. Il est prévu de porter SKY sur des transputers. Une étude est en cours de réalisation pour combiner SKY et d'autres algorithmes de visualisation disponibles à l'EMSE pour acquérir un outil de création d'images mixtes. Cet outil devrait permettre de visualiser rapidement des scènes en mélangeant des parties d'images calculées à l'aides d'algorithmes différents, ainsi le tracé de rayons ne serait utilisé que pour les objets réfléchissants ou réfractants tandis que les objets diffus pourraient être calculés avec un algorithme de type Atherton.

Annexe A

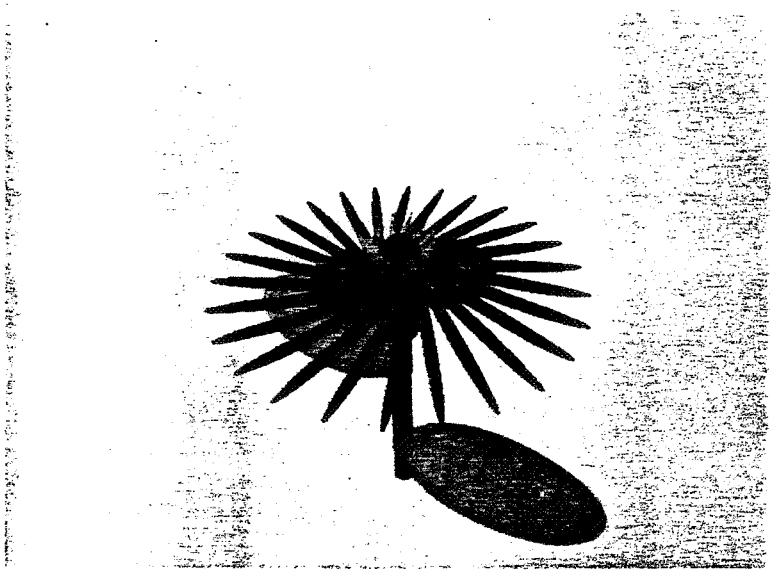


Image 1: Fleur aliassée

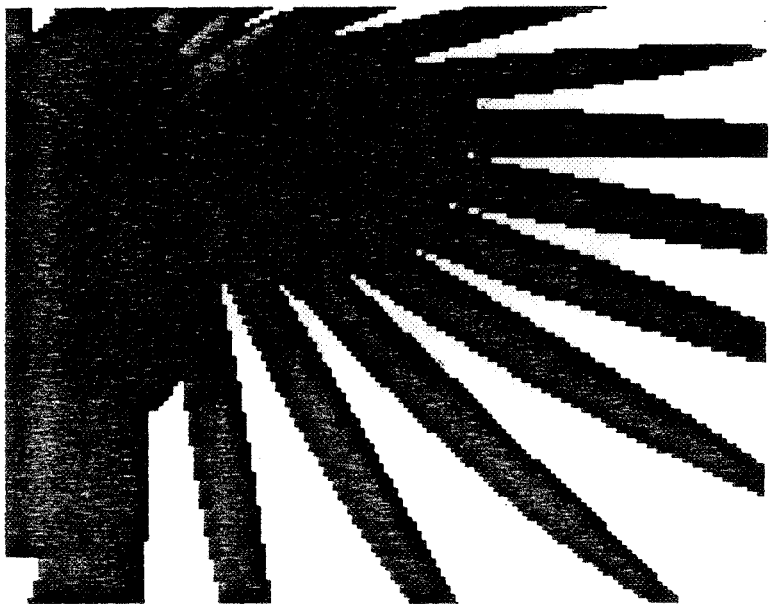


Image 2: Zoom sur fleur aliassée

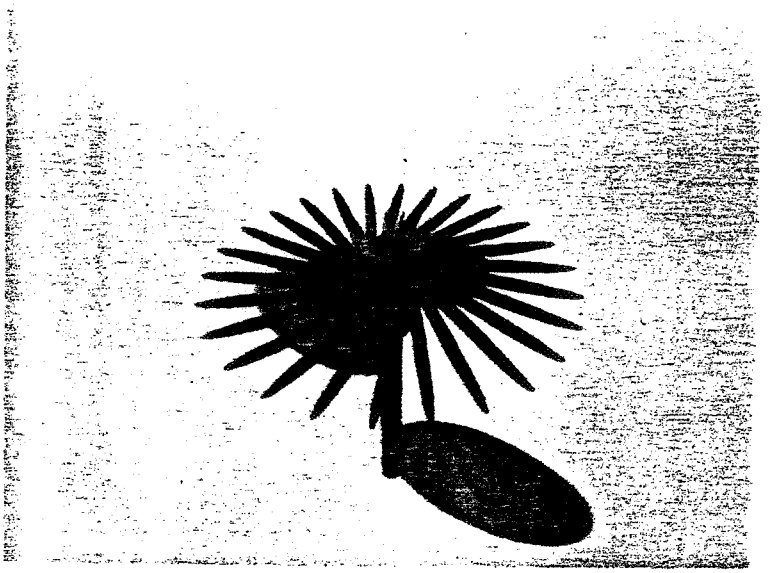


Image 2: Fleur antialiassée (cf Chapitre 3)

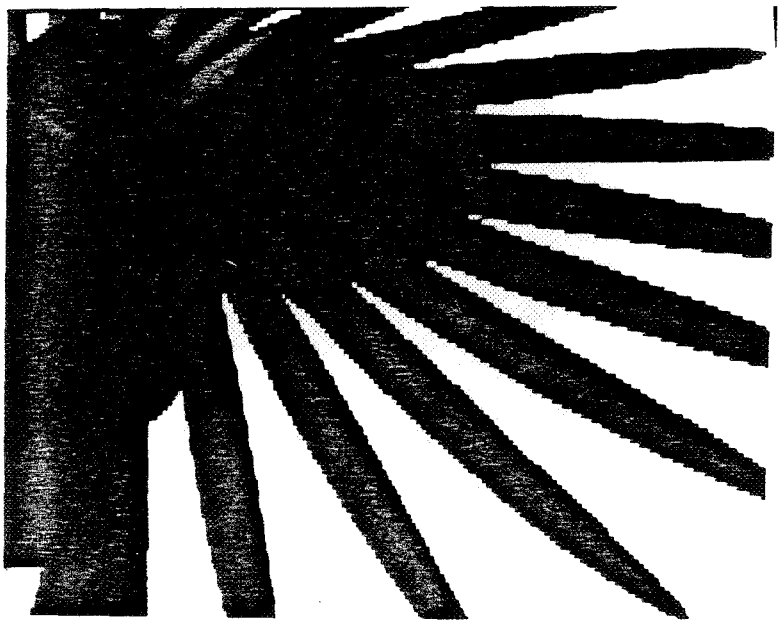


Image 4: Zoom sur fleur antialiassée

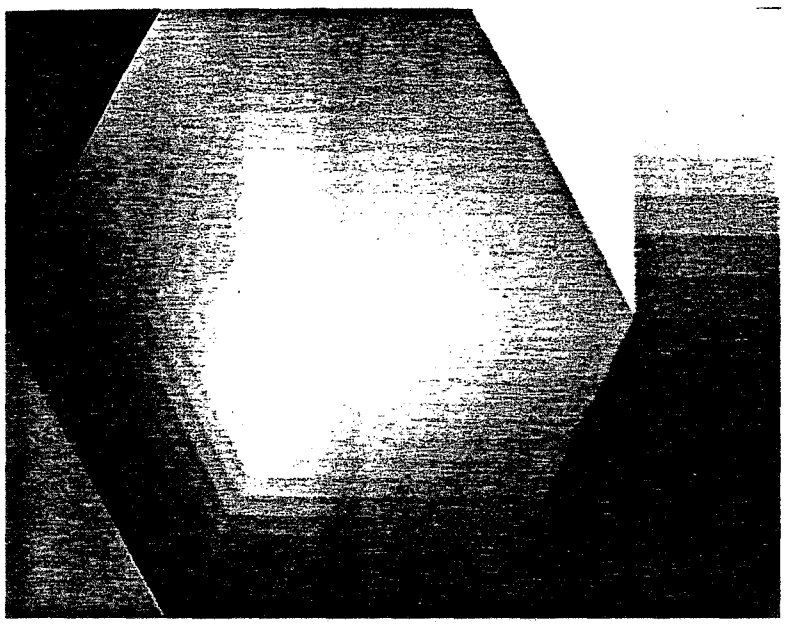


Image 5: Cube des couleurs en table fixe, 4096 couleurs (cf Chapitre 4).

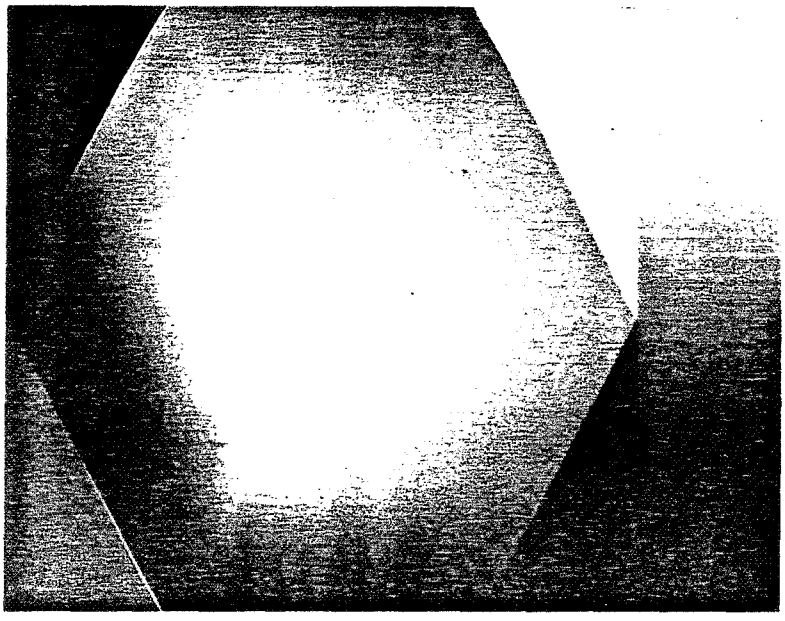


Image 6: Cube des couleurs en table variable, 4096 couleurs, table variable calculée par quantification des couleurs par un octree en suivant le balayage de Peano.

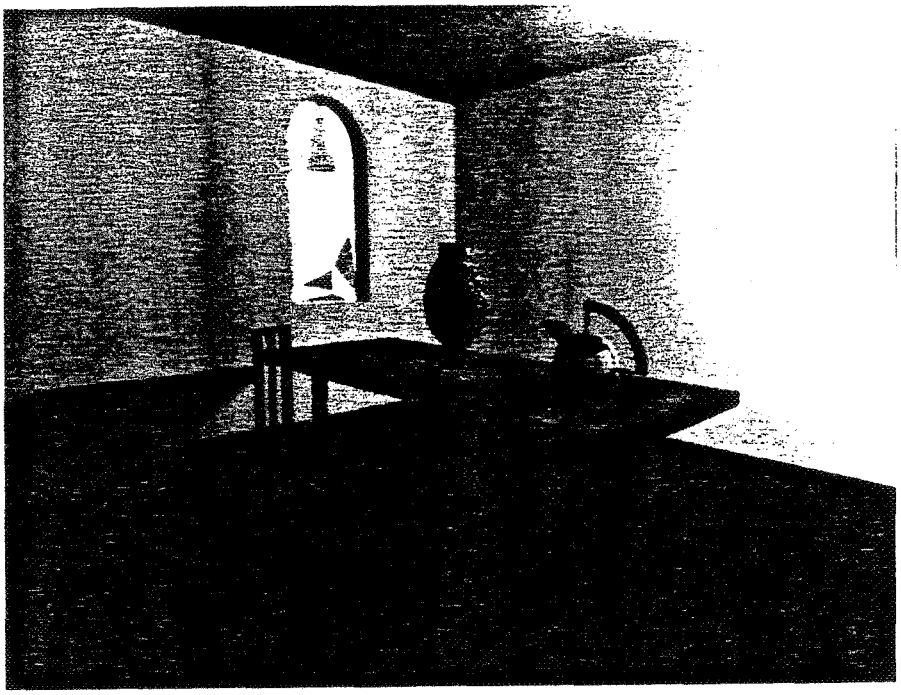


Image 7: La pièce au vitrail. Cette pièce est formée d'un cube creux (cf Chapitre 2). Les statistiques de cette image sont exposées dans l'annexe B.

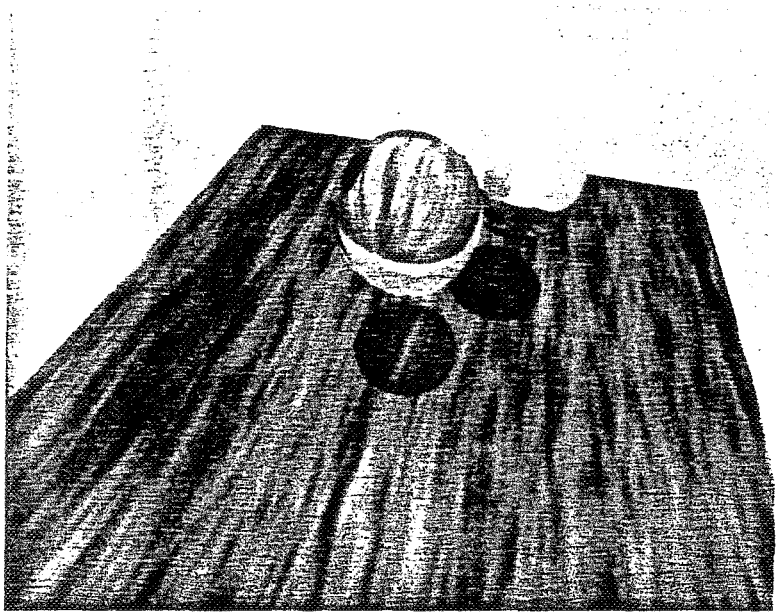


Image 8: Réflexions / réfractions.

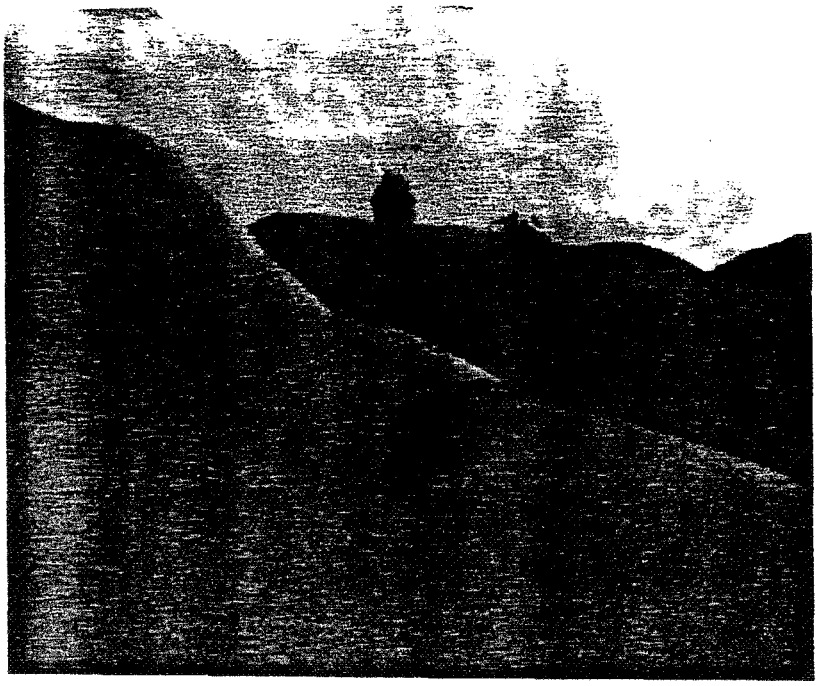


Image 9: Arbres 3D générés par une méthode spectrale. Terrain et ciel calculés par l'algorithme de Coquillart.

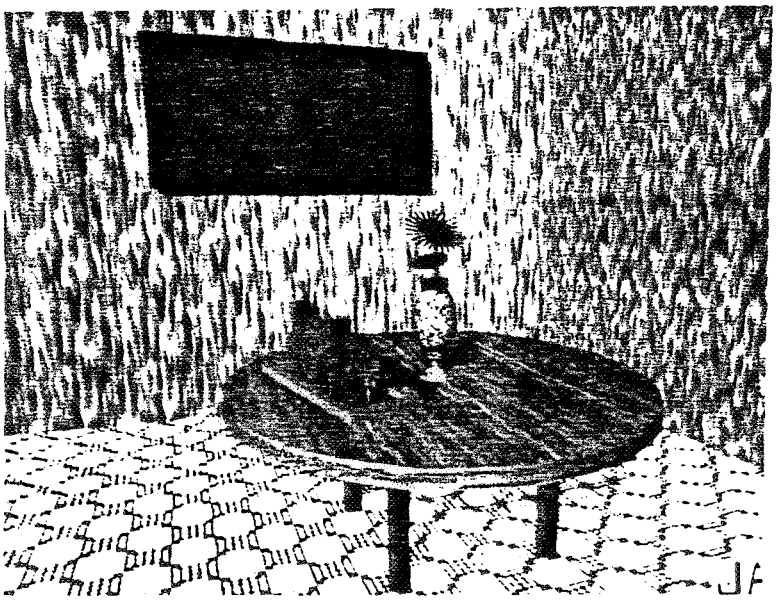


Image 10: "Ceci n'est pas une texture"

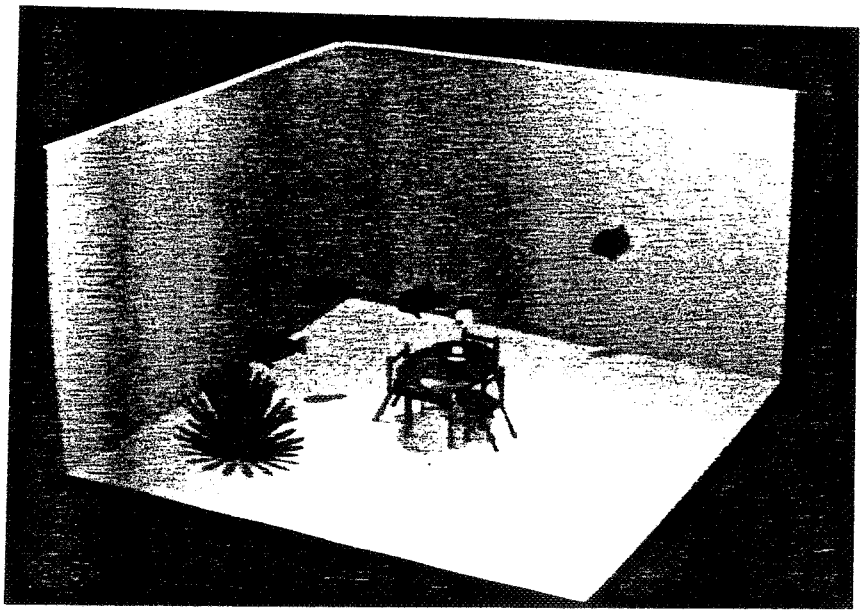


Image 11: Tea for sea. Conception Frank Chopin.

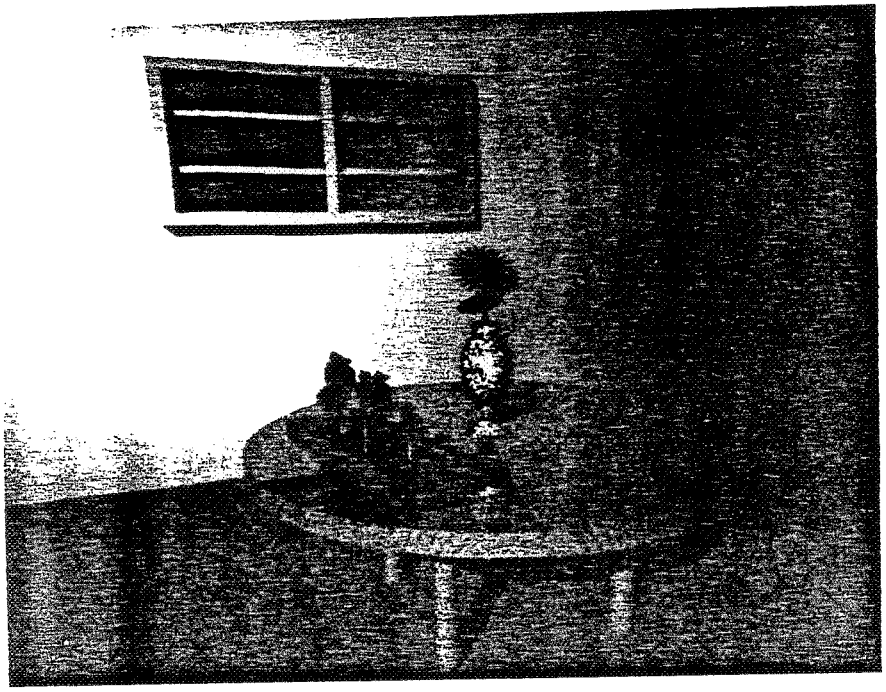


Image 12: Une pièce.

Annexe B

Le programme SKY contient environ 25 000 lignes de programme, dont 2 500 lignes pour SET_UP, 2800 pour KEEP et 19 700 pour YIELD.

Les statistiques suivantes portent toutes sur l'image 7. La scène est composées de 64 primitives. Elle comporte deux sources lumineuse, des réflexions, des réfractions, des différences, des intersections et une primitive creuse. Les temps suivants sont calculés sur une image de taille 250x250 sur un SPS90.

Caractéristiques de l'image :

Nombre de rayons primaires :	62 500
Nombre de rayons d'ombre pour les rayons primaires :	65 536
Nombre de rayons d'ombre pour les rayons secondaires :	2 862
Nombre de rayons de réflexion ou de réfraction :	14 508

Nombre de tests	**t**	**Nombre d'intersections:**	**i**	**rapport t/i**
primaires :	215 601	primaires :	118 303	1,822
d'ombre pour les rayons primaires :	637 231	d'ombre pour les rayons primaires :	201 638	3,160
d'ombre pour les rayons secondaires :	20 686	d'ombre pour les rayons secondaires :	4 473	4,624
de réflexion ou de réfraction :	143 641	de réflexion ou de réfraction :	14 508	9,900

La relative mauvaise performance de la partie réflexion/réfraction provient de ce que l'algorithme codé n'utilise pas le fait que dans une cellule du quadtree les objets sont triés. S'il utilisait ce fait la moitié des tests, en moyenne, pourraient être supprimés. On peut remarquer que, sans optimisation, le rapport **t/i** vaudrait 64.

Temps de calcul des rayons primaires selon la taille minimum de découpage de la zone écran:

1x1 pixel	**2x2 pixels**	**3x3 pixels**	**4x4 pixels**	**5x5 pixels**
6:06.5	6:11.0	6:14.6	6:25.0	6:28.4

Evidemment, la taille 1x1 est optimale pour les temps de calculs. Mais, si la place mémoire est limitée, l'augmentation le taille de la zone indivisible de l'écran permet

d'économiser de la place mémoire tout en gardant des temps relativement bons. Quand le nombre de primitives est élevé (>1000) et que celles-ci occupent beaucoup de surface sur l'écran, le temps d'initialisation peut handicaper les performances du programme, dans ce cas aussi, il convient de diminuer le découpage de l'écran et donc d'augmenter la taille de la zone indivisible de l'écran.

Temps de calculs selon la profondeur des quadtrees pour l'optimisations des rayons de réflexions/réfractions.

1	**2**	**3**	**4**	**5**
30:29.0	29:44.6	28:50.6	28:22.6	28:57.2

L'utilisation d'un cube creux apporte les performances suivantes :

Type de scène	**Temps d'initialisation**	**Temps de calculs**
Cube creux, 64 primitives dans la scène	6.5	6:08.4
Différence, 65 primitives dans la scène	6.8	12:30.5

Parallélisation de SKY:

La parallélisation de SKY donnes comme temps de calculs pour une image 100x100 comportant 16 primitives selon le nombre de tâches CAP:

1	2	3	4	5	6	7	8
28.8	15.4	11.3	9.4	8.1	7.2	6.5	6.4

Pour un nombre de tâches CAP égal à 8, on arrive aux limites amenées par les temps de lecture, d'initialisation et de distribution de la scène. Cette même image calculée en 512x640 sur un SUN3/110 avec 4 Megaoctets de mémoire centrale prend 546.4 s, sur CAPITAN avec 8 tâches CAP : 76.6 secondes, ce qui donne un gain d'environ 7.52.

Bibliographie

[AMAN84] AMANATIDES John. Ray tracing with cones. *Proc SIGGRAPH'84,* Computer Graphics, 18(3), July 1984, 129-135.

[AMAN87] AMANATIDES John, WOO Andrew. A fast voxel traversal algorithm for ray tracing. *Proc. EUROGRAPHICS*, North Holland, 1987, 3-10.

[ANDE82] D. P. ANDERSON, Hidden line elimination in projected grid surfaces, *ACM Transactions on Graphics*, 1 (4), october 1982.

[AONO84] AONO M. and KUNII T.L. Botanical trees image generation. *Computer Graphics and Applications,* 4(5), 1984, 10-33.

[APPE68] A. APPEL, Some techniques for shading machine renderings of solids, *SJCC*, 1968, 37-45.

[ARGE85] ARGENCE Jacqueline. Synthèse d'éléments naturels : arbres, *Rapport de DEA informatique,* Ecole des Mines de Saint-Etienne, juin 1985.

[ARGE86] ARGENCE Jacqueline. Tracé de rayons, un peu mieux, un peu plus vite, *Rapport Interne,* Ecole des Mines de Saint-Etienne.

[ARGE88] ARGENCE Jacqueline. Antialiasing for ray tracing using CSG modeling, *Proc. CG International'88,* New Trends in Computer Graphics, N. Magnenat-Thalmann and D Thalmann eds., Springer Verlag, 1988, 199-208.

[ARNA87] ARNALDI Bruno, PRIOL Thierry, BOUATOUCH Kadi. A new space subdivision method for ray-tracing CSG modelled scenes.*Visual Computer,* 3, 1987, 98-108.

[ARVO86] J. ARVO, Backward ray tracing, Siggraph 86 course #12, 1986.

[ARVO87] ARVO James, KIRK David. Fast ray tracing by ray classification. *Proc SIGGRAPH,* Computer graphics, 21(4), 1987, 55-64.

[ATHE83] P. ATHERTHON, A scan-line hidden surface removal procedure for constructive solid geometry, *Computer Graphics*, 17 (3), july 1983, 73-82.

[BARR84] BARR Alan H. Global and local deformations of solid primitives. *Proc. SIGGRAPH'84,* Computer Graphics, 18(3), July 1984, 21-30.

[BARR86] BARR Alan H. Ray tracing deformed surfaces. *SIGGRAPH'86,* Computer Graphics, 20(4), August 1986, 287-295.

[BEIG88] BEIGBEDER Michel. Un développement pour la modélisation et la visualisation en synthèse d'images : CASTOR, Thèse de doctorat, Ecole des Mines de Saint-Etienne, avril 1988.

[BENT79] BENTLEY J.L. and OTTMANN T., Algorithm for reporting and counting geometric intersections, *IEEE Transactions on Computers,* 28(9), 1979, 643-647.

[BENT80] BENTLEY J.L. and WOOD D., A optimal worst case algorithm for reporting intersections of rectangles, *IEEE Transactions on Computers,* 29 (7), july 1980, 571-576.

[BIER86] BIER E.A. and SLOAN K.R., Two-part texture mapping, *IEEE Computer Graphics and Applications,* september 1986, 40-53.

[BLIN78] J. F. BLINN, Simulation of wrinkled surfaces, *Computer Graphics*, 12 (3), august 1978, 286-292.

[BOUA87] BOUATOUCH Kadi, MADANI M.O., PRIOL Thierry, ARNALDI Bruno. A new algorithm of space tracing using a CSG model. *Proc. EUROGRAPHICS*, North Holland, 1987, 65-78.

[BOUA88] BOUATOUCH K and PRIOL T. Parallel Space tracing : an experience on an iPSC hypercube, *Proc. CG International'88,* New Trends in Computer Graphics, N. Magnenat-Thalmann and D Thalmann eds., Springer Verlag, 1988, 170-188.

[BOUV84] BOUVILLE Ch., BRUSQ R., DUBOIS J.L., MARCHAL I. Synthèse d'images par lancer de rayon : algorithmes et architecture. *Proc. CESTA*, Biarritz, Mai 1984, 683-696.

[BOUV84] BOUVILLE Ch., DUBOIS J.L., MARCHAL I. Generating high quality pictures by ray-tracing. *Proc. EUROGRAPHICS'84*, Copenhagen, September 1984, 161-177.

[BOUV85] BOUVILLE Christian. Bounding ellipsoids for ray-fractal intersection. *Proc. SIGGRAPH'85,* Computer Graphics, 19(3), July 1985, 45-52.

[BROC86] BROCK P. J., POLINSKY A. J., SLIVKA R. and GREENBERG D. P. Unified interactive geometric modeller for simulating highly complex environments. *Computer Aided Design,* 18(10), December 1986, 539-545.

[BRON84] BRONSVOORT Willem F., VAN WIJK Jarke J., JANSEN Frederik W. Two methods for improving the efficiency of ray casting in solid modelling, *Computer Aided Design,* 16(1), January 1984, 51-55.

[BRON85] BRONSVOORT Willem and KLOK Fopke. Ray tracing generalized cylinders. *ACM transaction on Graphics,* 4(4), October 1985, 291-303, corrigendum, *ACM transaction on Graphics,* 6(3), July 1987, 238-239.

[BRON86] BRONSVOORT Whillem F. Techniques for reducing boolean evaluation time in CSG scan-line algorithms. *Computer Aided Design,* 18(10), December 1986, 533-538.

[BU 87] BU Jichun, DEPRETTERE Ed F. A VLSI system architecture for high-speed radiative transfer 3D image synthesis. *Proc. EUROGRAPHICS*, North Holland, 1987, 221-234.

[CAPI86] CAPITAN : Hardware summary description, Matra Espace, n^0 EPT /DI/042/GG/TT, August 1986ù.

[CAPI87] CAPITAN : Software summary description, Matra Espace, n^0 EPT /DI/120/PI/TT, March 1987.

[CARE85] CAREY Richard J. and GREENBERG Donald P. Textures for realistic image synthesis. *Computer Graphics in Architecture,* 9(2), 1985, 125-138.

[CATM74] CATMULL E. E., A subdivision algorithm for computer display of curved surfaces, Ph.D. Thesis, University of Utah, Computer Science Department, Salt Lake City, Utah, 1974.

[CHAT87] CHATTAPADHYAY S. and FUJIMOTO A. Bi-directional ray tracing. *Computer Graphics,* 1987, TL KUNII ed, Springer Verlag, Tokyo, 335-343.

[CLEA86] CLEARY J. G., WYVILL B. , BIRTWISTLE G. M., and VATTI R. Multiprocessor ray tracing. *Computer Graphics Forum*, 5, 1986, 3-12.

[CLEA88] CLEARY J. G. and WYVILL G. Analysis of an algorithm for fast ray tracing using uniform space subdivision. *Visual Computer*, 4, 1988, 65-83.

[COHE85] COHEN M.F.,GREENBERG D.P. The hemi-cube : a radiosity solution for complex environments, *Computer Graphics*, 19 (3), july 1985, 31-40.

[COHE86] COHEN M.F., GREENBERG D.P., IMMEL D. S. and BROCK P. J. An efficient radiosity approach for realistic image synthesis, *IEEE Computer Graphics and Applications*, march 1986.

[COOK84] COOK R. L., PORTER T. and CARPENTER L., Distributed ray tracing, *Proc. SIGGRAPH'84,* Computer Graphics, 18 (3), july 1984, 137-145.

[COQU84] COQUILLART S., Représentation de paysages et tracé de rayon, Thèse, Ecole des Mines de St Etienne, 1984.

[COQU85] COQUILLART S., An improvement of the ray tracing algorithm, *Eurographics 85*, september 1985, 77-88.

[CORD85] CORDONNIER E., BOUVILLE C., DUBOIS J.L., MARCHAL I. Creating CSG modelled pictures for ray tracing display. *Proc. EUROGRAPHICS*, 1985, 171-182.

[CROW84] CROW F.C. Summed-area tables for textures mapping *Proc. SIGGRAPH*, Computer Graphics, 19(3), july 1984, 207-212.

[DEGU84] DEGUCHI H., NISHIMURA H., YOSHIMURA H., KAWATA T., SHIRAKAWA I., and OMURA K. A parallel processing scheme for three-dimensional image generation. *Proc. Int'l Symp. Circuits And Systems* , 1984, 1285-1288.

[DIPP84] DIPPE Mark, SWENSEN John. An adaptive subdivision algorithm and parallel architecture for realistic image synthesis. *Proc SIGGRAPH'84,* Computer graphics, 18(3), July 1984, 149-157.

[DIPP85] DIPPE M. A. and WOLD E. H. Antialiasing through stochastic sampling, *SIGGRAPH'85*,Computer Graphics, 19 (3), july 1985, 69-78.

[ENGL88] ENGLAND Nick. Application acceleration: development of the TAAC-1. *Sun Technology* , Winter 1988, 34-41.

[EXCO87] EXCOFFIER T., TOSAN E. Une méthode d'optimisation du lancer de rayon (Ray Casting). *Proc. MICCAD,* Hermès, 1987, 549-563.

[EYRO86] EYROLLES G., FRANCON J. et VIENNOT G. Combinatoire pour la synthèse d'images de plantes. *Proc. CESTA'86,* 2, Avril 86, 648-652.

[FORG87] FORGUE M-C, GIRAUDON G., BOERI F. et AUGUIN M. Parallelisation du lancer de rayons. *Proc CESTA,* Paris, Mai 1987, 57-62.

[FOUR82] FOURNIER A. , FUSSEL D. and CARPENTER L. Computer rendering of stochastic models, *Communications of A.C.M.*, 25, june 1982, 371-384.

[FUCH80] FUCHS H., REDEM Z.M. and NAYLOR B.F. On visible surface generation by a priori tree structure. *ACM,* April 1980, 124-133.

[FUJI86] FUJIMOTO A., TANAKA T and IWATA K. ARTS:Accelerated ray-tracing system. *Computer Graphics and Applications,* 6(4), 16-26.

[GAMS87] Groupe d'études pour l'Application des Méthodes Scientifiques à l'Architecture et l'Urbanisme. Rapport d'activité 1986-1987, Marseille, 1987.

[GANG84] GANGNET M. et MICHELUCCI D. Un outil graphique interactif, rapport 84-12, Ecole des Mines de St Etienne, octobre 1984.

[GARD84] GARDNER Geoffrey Y., Simulation of natural scenes using textured quadric surfaces, *Proc. SIGGRAPH'84,* Computer Graphics, 18 (3), july 1984, 11-20.

[GARD85] GARDNER Geoffrey Y., Visual simulation of clouds, *Proc. SIGGRAPH'85,* Computer Graphics, 19 (3), july 1985, 297-303.

[GERV86] GERVAUTZ Michael. Three improvements of the ray tracing algorithm for CSG trees. *Computers & Graphics*, 10(4), 1986, 333-339.

[GERV88] GERVAUTZ M. and PURGATHOFER W. A simple method for color quantization : octree quantization, *Proc. CG International'88,* New Trends in Computer Graphics, N. Magnenat-Thalmann and D Thalmann eds., Springer Verlag, 1988, 219-231.

[GHAZ85] D. GHAZANFARPOUR, Synthèse d'images et antialiassage, Thèse de docteur Ingénieur, Ecole des Mines de St Etienne, novembre 85.

[GLAS84] GLASSNER Andrew S. Space subdivision for fast ray tracing. *IEEE Computer Graphics and Applications*, 4(10), 1984, 15-22.

[GLAS88] GLASSNER Andrew S. Spacetime ray tracing for animation. *IEEE Computer Graphics and Applications,* 1988, 60-70.

[GOLD71] GOLDSTEIN R. A. and NAGEL R. 3-D visual simulation, *Simulation*, 16 (1), january 1971, 25-31.

[GOLD87] GOLDSMITH Jeffrey, SALMON John. Automatic creation of object hierarchies for ray tracing. *IEEE Computer Graphics and Applications*, May 1987, 14-20.

[GORA84] GORAL C. M., TORRANCE K.E., GREENBERG D.P. and BATTAILE B. Modeling the interaction of light between diffuse surfaces, *Computer Graphics*, 18 (3), 213-222, july 1984.

[GREE86] GREENBERG D.P., COHEN M.F. and TORRRANCE K.E. Radiosity : a method for computing global illumination. *Visual Computer*, 2, 1986, 291-297.

[GUO 87] GUO B., YAMAMOTO T. and AOKI Y. Stacked frame buffer algorithm for fast recovery solid synthesis process. *Computer Aided Design* , 19(7), September 1987, 338-346.

[HAIN86] HAINES Eric A., GREENBERG Donald P. The light buffer : a shadow-testing accelarator. *IEEE Computer Graphics and Applications,* 6(9), 1986, 6-16.

[HALL83] HALL Roy and GREENBERG Donald P. A testbed for realistic image synthesis. *IEEE Computer Graphics and Applications,* 3(8), November 1983, 10-20.

[HANR83] HANRAHAN Pat. Ray tracing algebraic surfaces. *Proc. SIGGRAPH'83,* Computer Graphics, 17(3), July 1983, 83-90.

[HANR86] HANRAHAN Pat. Using caching and breadth-first search to speed up ray-tracing. *Proc Graphics Interface,* Vancouver, 1986, 56-61.

[HECK80] HECKBERT P. Color image quantization for frame buffer display. B.S. Thesis, Architecture Machine Group, MIT, Cambridge Mass., 1980.

[HECK82] HECKBERT Paul, Color frame quantization for frame buffer display, *Proc. SIGGRAPH'82,* Computer Graphics, 16 (3), july 1982, 297-307.

[HECK84] HECKBERT P.S. Beam tracing polygonal objects, *Proc. SIGGRAPH'84*, Computer Graphics, 20(4), 1986, 119-127.

[HECK87] HECKBERT Paul S. Ray tracing JELL-O® brand gelatin. *Proc. SIGGRAPH'87,* Computer Graphics, 21(4), July 1987, 73-74.

[HOOK86] VAN HOOK T., Real-time shaded nc milling display, *Computer Graphics*, 20 (4), august 1986, 15-20.

[IMME86] IMMEL D. S. , COHEN M. F. and GREENBERG D. P. A radiosity method for non-diffuse environments, *Computer Graphics*, 20 (4), august 1986, 133-142.

[INAK88] INAKAGE M. Particals : an artistic approach to fuzzy objects, *Proc. CG International'88,* New Trends in Computer Graphics, N. Magnenat-Thalmann and D Thalmann eds., Springer Verlag, 1988, 126-134.

[JOY 86] JOY Keneth I. and BHETANABHOTLA Murthy N. Ray tracing parametric surface patches utilizing numerical techniques and ray coherence. *Proc. SIGGRAPH'86* , Computer Graphics, 20(4), August 1986, 279-285.

[KAJI82] KAJIYA James T. Ray tracing parametric patches. *Proc. SIGGRAPH'82,* Computer Graphics, 16(3), July 1982, 245-254.

[KAJI83] KAJIYA James T. New techniques for ray tracing procedurally defined objects. *Proc. SIGGRAPH'83*, Computer Graphics, 17(3), July 1983, 91-102.

[KAJI83] KAJIYA James T. Siggraph83 tutorial on ray tracing. Detroit.

[KAJI84] KAJIYA James T. and VON HERZEN Brian P. Ray Tracing volume densities. *Proc SIGGRAPH'84,* Computer Graphics, 18(3), July 1984, 165-173.

[KAWA82] KAWAGUCHI Y. A morphological study of the form of nature. *Proc. SIGGRAPH'82,* Computer Graphics, 16(3), 1982, 223-232.

[KAY 86] KAY Timothy L., KAJIYA James T. Ray tracing complex scenes. *Proc SIGGRAPH'86*, 20(4), August 1986, 269-277.

[KIRK86] KIRK David B. The simulation of natural features using cone tracing. *Advanced Computer Graphics*, TL KUNII ed., Springer Verlag, Tokyo, 1986, 129-143, or in *Visual Computer*, 3, 1987, 63-71.

[KOBA87] KOBAYASHI H., NAKAMURA T. and SHIGEI Y. Parallel processing of an object space for image synthesis using ray tracing.*Visual Computer*, 3, 1987, 13-22.

[KOBA88] KOBAYASHI H., NAKAMURA T. and SHIGEI Y. A strategy for mapping parallel ray-tracing into hypercube multiprocessor system, *Proc. CG International'88,* New Trends in Computer Graphics, N. Magnenat-Thalmann and D Thalmann eds., Springer Verlag, 1988, 160-169.

[KUNI85] KUNII Tosiyasu L., WYVILL Geoff. CSG and ray tracing using functional primitives. *Computer Generated Images*, The state of the art, Magnenat-Thalmann N, Thalmann D eds, Springer Verlag, Tokyo, 1985, 137-152.

[LANE79] LANE J.M. and CARPENTER L. C. A generalized scan line algorithm for the computer display of parametrically defined surface, *Computer Graphics and Image Processing*, 11, 1979, 290-297.

[LAUR85] LAURINI R., Graphical data bases built on Peano space-filling curves, *Proc. EUROGRAPHICS'85*, september 1985, 327-338.

[LAUR86] LAURINI R. et MILLERET F., Les clés de Peano en synthèse d'images : modélisation et opérateurs, *Proc. CESTA'86*, Nice, avril 1986, 659-667.

[LEE85] LEE M. E., REDNER R. A. and USELTON S. P., Statistically optimized sampling for distributed ray tracing, *SIGGRAPH'85*, Computer Graphics, 19 (3), july 1985, 61-67.

[LEHA84] LEHAR A.F. and STEVENS R.J., High-speed manipulation of the color chromaticity of digital images, *IEEE Computer Graphics and Applications*, february 1984, 34-39.

[LEIS88] LEISTER W., MAUS Th., MÜELLER H., NEIDECKER B. and STÖSSER A. "Occursus cum novo", computer animation by ray tracing in a network, *Proc. CG International'88*, New Trends in Computer Graphics, N. Magnenat-Thalmann and D Thalmann eds., Springer Verlag, 1988, 83-92.

[LIEN87] LIENHART P. et FRANCON J. Synthèse d'umages de feuilles végétales. *Proc. CESTA*, 2, Mai 1987, 212-218.

[MAGN88] MAGNENAT-THALMANN Nadia and THALMANN Daniel. Image synthesis - theory and practice. *Springer Verlag*, Tokyo, 1987.

[MAND77] MANDELBROT B. Fractals: form , chance and dimension, ed. W.H. Freeman and co, San Francisco, 1977.

[MANT83] MANTYLA M. and TAMMINEN M. Localized set operations for solid modeling, *Computer Graphics*, 17 (3), july 1983, 25-29.

[MARI86] MARINI Daniele and TAMPIERI Filippo. A ray-tracing system for image synthesis. *Proc. CESTA*, Nice, Avril 1986, 810-816.

[MAST87] MASTIN G.A., WATTERBERG P.A. and MAREDA. Fourier synthesis of ocean scenes. *Computer Graphics and Applications*, 7(3), 16-23.

[MAXW86] MAXWELL G. M., BAILEY M. J. and GOLDSCHMIDT V. W. Calculations of the radiation configuration factor using ray casting. *Computer Aided Design*, 18(7), September 1986, 371-379.

[MEYE86] MEYER Gary W., Tutorial on color science, *Visual Computer*, 2, 1986, 278-290.

[MEYE88] MEYER Bertrand, Object-oriented software construction, *Prentice Hall International*, Cambridge, 1988.

[MICH87] MICHELUCCI D., Les représentations par les frontières : quelques constructions ; difficultés rencontrées, Thèse, Ecole des Mines de St Etienne, Novembre 1987.

[MITC87] MITCHELL Don P. Generating antialiased images at low sampling densities, *Proc. SIGGRAPH'87*, Computer Graphics, 21(4), july 1987, 65-72.

[MOIS84] MOISSINAC J.C., Aides informatiques à la réalisation de dessins animés, Thèse, Ecole des Mines de St Etienne, 1984.

[MOOR77] MOORE R. E. and JONES S. T., Safe starting regions for iterative methods, *SIAM J. Numerical Analysis*, 14 (6), december 1977, 1051-1065.

[NARU87] NARUSE T., YOSHIDA M., TAKAHASHI T. and NAITO S. Sight - A dedicated computer graphics machine. *Computer Graphics Forum*, 6(4), 1987, 327-334.

[NEMO86] NEMOTO Keiji, OMACHI Takao. An adaptive subdivision by sliding boundary surfaces for fast ray tracing. *Proc. Graphics Interface*, Vancouver, 1986, 43-48.

[NIER82] NIERVEGELT J. and PREPARATA F.P. Plane sweep algorithm for intersecting geometric figures. *Communication of the ACM*, 25(10), October 1982, 739-747.

[NISH83] NISHIMURA H., OHNO H., KAWATA T., SHIRAKAWA I., and OMURA K. Links-1: a parallel pipelined multimicrocomputer system for image creation. *Proc SIGARCH'83*, ACM,New York, 1983, 387-394.

[NISH85] NISHITA T. and NAKAMAE E., Continuous tone representation of three-dimmensional objects taking account of shadows and interreflection, *Computer Graphics*, 19 (3), july 1985, 23-30.

[NISH86] NISHITA T. and NAKAMAE E., Continuous tone representation of three-dimmensional objects illuminated by sky light, *Computer Graphics*, 20 (4),, august 1986, 125-132.

[NORT82] NORTON A., ROCKWOOD A.P. and SKOLMOSKI P.T. Clamping: method of antialiasing textured surfaces by bandwidth limiting in object space. *Proc. SIGGRAPH'82*, Computer Graphics, 16(3), july 1982, 1-8.

[OHTA87] OHTA Masataka, MAEKAWA Mamoru. Ray Coherence Theorem and Constant Time Ray Tracing Algoritm. *Computer Graphics*, Tokyo, 1987, 303-314.

[PEAC86] PEACHEY Darwyn R. Portray-An image synthesis system. *Proc Graphics interface,* Vancouver, 1986, 37-42.

[PEAN1890] PEANO G. Sur une courbe qui remplit toute une aire plane. *Mathematische Annalen,* Bd XXXVI, 1890, 157-160.

[PENG87] PENG Qunsheng, ZHU Yining, LIANG Youdong. A fast ray tracing algorithm using space indexing techniques. *Proc. EUROGRAPHICS*, North Holland, 1987, 11-23.

[PERO88] PEROCHE B., ARGENCE J., GHAZANFARPOUR D. et MICHELUCCI M. La synthèse d'images. *Hermès,* Paris, 1988.

[PLUN85] PLUNKETT D. J., BAILEY M. J. The vectorization of a ray tracing algorithm for improved execution speed. *IEEE Computer Graphics and Applications,* 5(8), august 1985, 52-60.

[POUG87] POUGET-ARGENCE Monique. Présentation du logiciel de CAO : EUCLID, communications personnelles, 1987.

[PRUSI88] PRUSINKIEWICZ P., LINDENMAYER A. and HANA J. Developmental models of herbaceous plants for computing imagery purposes. *Proc. SIGGRAPH'88*, Computer Graphics, 22(4), August 1988, 141-150.

[PULL87] PULLEYBLANK Ron, KAPENGA John. The feasibility of a VLSI chip for ray tracing bicubic patches. *IEEE Computer Graphics and Applications*, , March 1987, 33-44.

[PURG86] PURGATHOFER Werner. A statistical method for adaptive stochastic sampling, *Eurographics 86*, august 1986, 145-152.

[PURG88] PURGATHOFER Werner and GROELER Eduard, Using tetrehedrons for dithering color pictures, *3rd International Conference on CG*, Dudrovnik, Yougoslavia, june 1988.

[RALS65] RALSTONE A., A first course in nûmerical analysis, Mc-Graw-Hill, 1965.

[REEV83] REEVES W.T. Particle systems - a technique for modeling a class of fuzzy objetcs. *Porc SIGGRAPH'83,* Computer Graphics, 17(3), July 1983, 359-376.

[REEV85] REEVES W.T. and BLAU R. Approximate and probabilistic algorithms for shading and rendering structured particles systems. *Proc. SIGGRAPH'85,* 19(3), July 1985, 313-322.

[REQU80] REQUICHA A.A.G. Representation for rigid solids : theory, methods and systems, *ACM Computing Surveys*, 12, december 1980, 110-116.

[REYF88] REYFFE P.D., EDELIN C., FRANCON J., JAEGER M. and PUECH C. Plant models faithful to botanical structure and development *Proc. SIGRRAPH'88,* 22(4), August 1988, 151-158.

[ROTH82] ROTH Scott D. Ray casting for modeling solids. *Computer Graphics and Image Processing*, 18, 1982, 109-144.

[RUBI80] RUBIN Stevens M., WHITTED Turner. A 3-dimensional representation for fast rendering of complex scenes. *Proc. SIGGRAPH*, Computer Graphics, 14(3), 1980, 110-116.

[SAME86] SAMEK M., SLEAN C. and WEGHORST H., Texture mapping and distortion in digital graphics, *Visual Computer*, 2, 1986, 313-320.

[SCHA80] SCHACHTER B.J. Long crested wava models. *Computer Graphics and Image Processing*, 12, 187-201.

[SCHE87] SCHERSON Isaac D., CASPARY Elisha. Data structures and the time complexity of ray tracing. *The Visual Computer*, 3, 1987, 201-213.

[SCHO86] SCHOELER Peter and FOURNIER Alain. Profiling graphics display systems, *Proc. Graphics Interface'86*, Canadian Information Procesing Soc., Toronto, 1986, 49-55.

[SCHW87] SCHWARTZ M. W., COWAN W. B. and BEATTY J.C. An experimental comparison of RGB, YIQ, LAB, HSV, and opponent colors models. *ACM Transaction on Graphics*, 6(2), April 1987, 123-158.

[SEDE84] SEDERBERG Thomas W. and ANDERSON David C. Ray tracing of Steiner patches. *Proc. SIGGRAPH'84*, Computer Graphics, 18(3), July 1984, 159-164.

[SHAM76] SHAMOS M.I. and HOEY D. Geometric intersections problems. *Proc. of the 17th IEEE Symposium on Fondamentions of Computing*, 1976, 208-215.

[SHIN87] SHINYA M., TAKAHASHI T. and NAITO S. Principles and application of pencil tracing. *Proc. SIGGRAPH'87*, Computer Graphics, 21(4), July 1987, 45-54.

[SHOU79] SHOUP R. G. Color table animation. *Computer Graphics*, 13(2), August 1979.

[SMIT84] SMITH A.R. Plants, fractals and formal languages. *Proc. SIGGRAPH'84*, Computer Graphics, 18(3), 1984, 1-9.

[SNYD87] SNYDER John and BARR Alan H. Ray tracing complex models. *Proc. SIGGRAPH'87*, Computer Graphics, 21(4), July 1987, 119-128.

[SORE82] P. SORENSEN, Tronic imagery, *Byte*, 7 (11), November 1982, 48-74.

[SPEE85] SPEER Richard L., DEROSE Tony D., BARSKY Brian A. A theorical and empirical analysis of coherent ray-tracing. *Computer Generated Images*, The state of the art, Magnenat -Thalmann N, Thalmann D eds, Springer Verlag, Tokyo, 1985,11-25.

[STEV83] STEVENS R.J., LEHAR A.F. and PRESTON F.H., Manipulation and presentation of multidimensional image data using the Peano scan, *IEEE Transactions on Pattern analysis and machine intelligence,* 5 (5), september 1983, 520-526.

[STRO86] STROUSTRUP Bjarne. The C++ programming language. Addison-Wesley, Reading, 1986.

[SUTH74] SUTHERLAND I. E., SPROULL R.F. and SCHUMACKER R. A. A characterization of ten hidden-surface algorithms, *Computing Surveys,* 6 (1), january 1974, 1-55.

[TAKA87] TAKAHASHI T., YOSHIDA M., NARUSE T. Architecture and performance evaluation of the dedicated graphics computer : SIGHT. *Proc. IEEE MONTECH'87-COMPINT'87,* November 1987, 153-160.

[TALL85] TALLOT Didier, Quelques propositions pour la mise en œuvre d'algorithmes combinatoires, Thèse, Ecole des Mines de St Etienne, juin 1985.

[TAMM82] TAMMINEN M., SULONEN R. The excell method for efficient geometric acces to data, *Proc. IEEE,* 19th design automation conference, 1982, 345-351.

[TAMM84] TAMMINEN M., KARONEN O. and MÄNTYLÄ M., Ray-casting and block model conversion using a spatial index, *Computer-Aided Design,* 16 (4), july 1984, 203-208.

[TEZE85] TEZENAS DU MONTCEL Bruno and NICOLAS Alain. An illumination model for ray-tracing. *Proc. MICCAD,* Hermès, 1987, 63-75.

[THOM86] THOMAS Spencer W. Dispersive refraction in ray tracing. *Visual Computer,* 2, 1986, 3-8.

[THOM87] THOMAS Michel. Réflexions et transparences dans le tracé de rayon SKY. Rapport de travail de fin d'études, Ecole des Mines de St Etienne, 1987.

[TOTH85] TOTH Daniel L. On ray tracing parametric surfaces. *Proc. SIGGRAPH'85,* Computer Graphics, 19(3), July 1985, 171-179.

[VERB84] VERBECK Channing P. and GREENBERG Donald P. A comprehensive light-source description for computer graphics. *IEEE Computer Graphics and Applications,* 4(7), July 1984, 66-75.

[VERR87] VERROUST A. Visualization algorithm for CSG polyhedral solids. *Computer-Aided Design,* 19(10), December 1987, 527-533.

[WALL87] WALLACE J.R., COHEN M.F. and GREENBERG D.P. A two-pass solution to the rendering equation : a synthesis of ray tracing and radiosity methods. *Proc. SIGGRAPH'87,* Computer Graphics, 21(4), july 1987, 311-320

[WARD88] WARD G.J., RUBINSTEIN F. M. and CLEAR R.D., A ray tracing solution for diffuse interreflexion, *Proc. SIGGRAPH'88,* Computer Graphics, 22 (4), august 1988, 85-92.

[WARN83] WARN D.R. Lighting controls for synthetic images, *Computer Graphics,* 17(3), July 1983.

[WATK70] WATKINS G. S., A real time visible surface algorithm, University of Utah, Computer Science Department, UTEC-CSc-70-101, june 1970.

[WEGH84] WEGHORST Hank, HOOPER Gary, GREENBERG Donald P. Improved computational methods for ray tracing. *ACM Trans. on Graphics,* 3(1), 1984, 52-69.

[WHIT80] WHITTED Turner. An improved illumination model for shaded display. *Comm. ACM* , 23(6), June 1980, 343-349.

[WIJK84] VAN WIJK Jarke J. Ray tracing object defined by sweeping planar cubic splines. *ACM transaction on Graphics,* 3(3), July 1984, 223-237.

[WILL72] WILLIAMSON H. Hidden line plotting algorithm, *Communications of the ACM*, 15 (2), february 1972, 100-103.

[WILL83] WILLIAMS L. Pyramidal parametrics, *Proc. SIGGRAPH'83,* Computer Graphics, 17(3), july 1983,

[WRIG73] WRIGHT T. J., A two-space solution to the hidden line problem for-plotting functions of two variables, IEEE Transaction on Computers, C-22 (1), january 73, 1-11.

[WYVI88] WYVILL G. and SHARP P. Volume and surface properties in CSG, *Proc. CG International'88,* New Trends in Computer Graphics, N. Magnenat-Thalmann and D Thalmann eds., Springer Verlag, 1988, 257-266.

[YAMR87] YAMROM Boris. Ray Tracing : An object-oriented Design. Siggraph 87 course #27, 1987.

[YANG87] YANG Chang-Gui. On speeding up ray tracing of B-spline surfaces. *Computer Aided Design,* 19(3), April 1987, 122-130.

Chapitre 1: Présentation générale de l'algorithme du tracé de rayons

Chapitre 2 : Optimisations

Chapitre 3: Antialiassage du tracé de rayons

Chapitre 4: Couleurs

Chapitre 5: SKY un logiciel de tracé de rayons

N° d'ordre : ID 21

Résumé :

Le tracé de rayons est à l'heure actuelle un des algorithmes permettant de produire les images les plus réalistes. Mais cette technique est très coûteuse en temps de calcul. Plusieurs nouvelles méthodes d'optimisation sont proposées. Une extension de la modélisation par arbre de construction a été développée. Pour améliorer la qualité des images sans trop augmenter les temps de calcul, un algorithme d'antialiassage rapide est présenté. Une méthode de construction d'une table de couleurs par quantification des couleurs dans un octree en utilisant le balayage de Peano est étudiée. Ces techniques ont été implantées au sein du logiciel SKY, porté sur une machine parallèle.

Mots clés :

Synthèse d'images, tracé de rayons, arbre de construction, optimisations, antialiassage, couleur, parallélisme.